PROYECTO FIN DE CARRERA

Diseño e implementación de aplicación web para validación remota de programas C

Autor: Pablo López Anastasio

Tutor: Pablo Basanta Val

Leganés, Febrero de 2012

Título: Diseño e implementación de aplicación web para validación remota de programas C

Autor: Pablo López Anastasio

Director: Pablo Basanta Val

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de _____ de 20__ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

SECRETARIO VOCAL PRESIDENTE

Agradecimientos

Agradezco a mi tutor, Pablo, su orientación y consejos para llevar a cabo el proyecto. También a Manuel Jesús Martín Gutiérrez por el buen trabajo que hizo, ya que su Proyecto de Final de Carrera me ha sido muy útil para guiarme.

Además también le doy las gracias a mi familia y a Alba porque, a pesar de no entender nada de lo que les contaba, siempre han estado a mi lado apoyándome en todo momento.

Por último agradecer a todos los compañeros y amigos que han estado conmigo estos cinco años de universidad porque han hecho que esta época de mi vida sea inolvidable.

Resumen

El proyecto que se va a realizar está orientado, en principio, al entorno docente. Consiste en dar al alumno la posibilidad de validar fácilmente sus programas en C a través de una plataforma web. Internamente se usa la herramienta de código libre Valgrind, que permite conocer, entre otras informaciones, las fugas de memoria y los problemas de concurrencia del código del usuario. Finalmente, para comodidad de éste, se envía el resultado obtenido en la plataforma al correo electrónico introducido en el formulario inicial.

Dentro del proyecto que se va a realizar se analizan las tecnologías que mejor se adaptan a él, como son Java EE en su versión 6, por ofrecer una infraestructura sencilla donde incluir el acceso a la plataforma y la gestión de peticiones, el servidor de aplicaciones Glassfish, por su fácil instalación y su buen rendimiento, y la suite de herramientas de depuración Valgrind, por ser una de las más completas en este campo.

Además se diseña la plataforma, incluyendo todos los módulos necesarios para su funcionamiento. Entre ellos se encuentran el módulo de acceso a la plataforma, el de gestión de la cola de peticiones, el de validación del código del usuario y el de notificación vía correo electrónico.

También se desarrolla la configuración y el código necesarios para hacer usable esta aplicación. Se implementa el módulo de acceso con JSP y servlets, la cola de peticiones con JMS y las clases Java encargadas de procesar el programa del usuario y la respuesta final.

Más adelante se llevan a cabo unas pruebas de rendimiento utilizando unos ejemplos de programas en código C obtenidos del *Lawrence Livermore National Laboratory* que comprueban la utilidad que tiene la plataforma.

Por último, se finaliza con unas conclusiones sobre todo el trabajo realizado y la viabilidad del proyecto y con unas líneas de trabajo futuro, entre las que se encuentran el acoplamiento del proyecto que se va a realizar con otro anterior y el desarrollo más avanzado de la aplicación en cuanto al resto de herramientas de Valgrind.

Palabras clave:
Valgrind, EJB, Java EE, JMS, Glassfish, aplicación web, código libre.

Abstract

This project aims, in principle, the learning environment. It consists on giving to students the opportunity to easily validate their C programs through a web platform. Internally it uses the open source tool Valgrind, which identifies, among other information, memory leaks and concurrency issues from user code. Finally, for the convenience of it, it sends the result obtained in the e-mail platform introduced in the initial form.

Within this project analyzes the technologies best suited to it, such as Java EE in version 6, to offer a simple infrastructure which includes access to the platform and management requests, the Glassfish application server, because of his easy installation and good performance, and the suite of debugging tools Valgrind, as one of the most comprehensive in this field.

Besides, designing the platform, including all modules necessary for its operation. These include the module platform access, the management of the request queue, the validation of the user code and notification via email.

It also develops the configuration and code needed to make this application usable. Access module is implemented with JSP and servlets access, the request queue with JMS and Java classes responsible for processing the user program and the final answer.

Below, it carries out a performance test using examples of C code programs obtained from the *Lawrence Livermore National Laboratory* which prove the usefulness of the platform.

Finally, it ends with some conclusions on all the work done and the project viability and future lines of work, among which are the coupling of this project with an earlier and more advanced development of the application as to other Valgrind tools.

Keywords:
Valgrind, EJB, Java EE, JMS, Glassfish, web application, open source.

Índice general

Apéndices

Índice de figuras

Índice de tablas

Capítulo 1

Introducción y objetivos

1.1 Introducción

La validación formal ([7]) aparece por la necesidad de analizar el funcionamiento de sistemas complejos que utilizan aplicaciones concurrentes. Esto implica que no es suficiente saber si dichas aplicaciones respetan el lenguaje en que se desarrollaron, sino que además hay que controlar que acceden a los recursos de forma que no se ponga en peligro el funcionamiento del sistema completo. En la vida real hay muchos sistemas de este tipo y los usamos en nuestro día a día, por eso es imprescindible garantizar su seguridad.

Estas aplicaciones pueden estar escritas en distintos códigos, tales como Java o C. El primero es un código de mayor nivel, e incluye entre sus características la capacidad de liberar memoria del equipo cuando ya no va a ser usada. Sin embargo, en las aplicaciones desarrolladas en C es el propio programador el que tiene que encargarse de hacer esa tarea. Por eso la validación de este código ha de incluir también la función de comprobar si la memoria utilizada por el programa en cuestión es tratada correctamente.

Ya existen herramientas que permiten validar programas escritos en ambos códigos. Para Java podemos encontrar Java Path Finder ([8]), desarrollada por la NASA, y para C están algunas como Valgrind ([6]) o MyGCC ([9]).

Partiendo de lo anterior, el proyecto que se va a realizar está orientado a ofrecer al usuario la posibilidad de validar su programa vía web sin necesidad de instalar en su equipo ninguna de las herramientas antes mencionadas. Para la validación remota de código Java, podemos encontrar ya una plataforma desarrollada por un alumno de la Universidad Carlos III de Madrid ([1]), pero para código C no se ha desarrollado un equivalente todavía. Por eso, el proyecto que se va a realizar pretende completar la tarea de ofrecer al usuario la validación remota de sus programas incluyendo una plataforma encargada de hacerlo para código C.

Para llevar a cabo esto, se empleará la tecnología Java EE ([10]) y el servidor de aplicaciones Glassfish ([5]). La razón es que ambas son escalables y portables entre plataformas. Sin embargo, esta aplicación tendrá que funcionar sobre un sistema operativo Linux, ya que actualmente la herramienta Valgrind no está disponible para otro.

1.2 Objetivos

El objetivo fundamental del proyecto es el de crear una aplicación utilizando Java EE que, integrada en un servidor, permita al usuario validar vía Web sus programas en código C. De esto se deducen los siguientes subobjetivos:

1. **Analizar la herramienta de validación Valgrind**

   Es una suite de herramientas de código libre bastante completa y por eso se ha empleado en el proyecto que se va a realizar. Además existen otras opciones que también han sido valoradas como alternativas viables.

2. **Analizar la última versión de Java EE y Glassfish**

   Sun Microsystems, ahora propiedad de Oracle ([24]), ofrece ambas tecnologías gratuitamente desde su sitio Web. Su uso permite que la aplicación desarrollada sea escalable y portable.

3. **Implementar la integración Valgrind – JavaEE**

   Es decir, implementar las clases e interfaces Java necesarias, así como la instalación y configuración del entorno donde se ejecutarán.

4. **Analizar la viabilidad y rendimiento del sistema completo**

   Realizar las pruebas necesarias para comprobar si el sistema es o no válido para su uso en un entorno docente real.

1.3 Fases del desarrollo

Para la realización del proyecto se han empleado aproximadamente seis meses. En este apartado se hace un repaso de las fases que se han seguido, así como el tiempo aproximado que se ha empleado en cada una de ellas.

1. **Estudio y revisión de proyectos similares (2 meses)**

   Se ha hecho un estudio de un proyecto equivalente al que aquí se desarrolla pero para validación de código Java. Se pensó inicialmente en realizar un módulo extra para el proyecto ya desarrollado, pero se optó al final por realizar la implementación desde cero, ya que era menos costoso debido a la estructura que tenía el proyecto.

2. **Estudio y elección de las posibles tecnologías a emplear (1 mes)**

   Se han estudiado varias tecnologías de las que luego se han seleccionado las que más se ajustaban al proyecto que se va a realizar concreto. Esas tecnologías son:

Java EE para el soporte y el desarrollo del código, Glassfish para la parte del servidor de aplicaciones, GCC como compilador de código C y Valgrind para las tareas de depuración.

3. Diseño de la plataforma (15 días)

Se ha diseñado la plataforma según el requisito principal: ofrecer al usuario la opción de validar sus programas en C vía web. Partiendo de ese punto se han diseñado los diferentes componentes y subcomponentes necesarios, como son el acceso web que permite al usuario utilizar la plataforma, la cola de peticiones para no saturar el servidor, el validador para procesar el código del usuario y el gestor de correos para informar del resultado.

4. Implementación (1 mes)

En esta fase se ha desarrollado el código necesario y se ha configurado el entorno para poder llevar a cabo el diseño. Ha sido necesario desarrollar tres JSPs y un *servlet* para la parte de acceso a la plataforma, un *Message-Driven Bean* con la configuración apropiada de *Java Messaging Service* y el servidor de aplicaciones para la gestión de la cola de peticiones y varias clases Java para el procesado del código del usuario y el envío del correo electrónico con el resultado.

5. Diseño y puesta en marcha de las pruebas de rendimiento (10 días)

Para llevar a cabo las pruebas se han utilizado diez programas en código C obtenidos del *Lawrence Livermore National Laboratory* que tienen diferentes características relacionadas con el uso de la memoria y de la sincronización. Han permitido comprobar que la plataforma desarrollada es viable y qué tipo de resultados se pueden obtener.

6. Redacción de la memoria (1 mes)

En esta fase se ha redactado toda la memoria del proyecto utilizando la documentación obtenida a lo largo de los meses anteriores y el código junto con la configuración de plataforma desarrollada.

7. Preparación de la presentación (7 días)

En esta última fase se ha preparado una presentación con diapositivas para la exposición del proyecto ante el tribunal y se ha ensayado varias veces para conseguir que dicha exposición sea completa y correcta.

1.4 Medios utilizados

Hardware

- **Ordenador portátil con conexión a Internet**

El hecho de que el ordenador sea portátil no es relevante, ya que no se ha necesitado tener movilidad. Podría haberse hecho también con un ordenador de sobremesa. Sin embargo, la conexión a Internet ha sido imprescindible para descargar el software necesario, buscar información y mantener el contacto con el tutor del proyecto. Para probar la plataforma en local no ha sido necesaria, pero para probarla remotamente también ha sido imprescindible la conexión a Internet.

- **Pantalla auxiliar**

Ha servido para agilizar la fase de implementación, ya que disponiendo de dos pantallas es más rápido comparar código y ver el resultado de ciertos cambios en alguna parte del programa.

Software

- **SO Linux: Ubuntu 11.04 ([2])**

Era necesario llevar a cabo el proyecto en el sistema operativo Linux, ya que Valgrind solo está disponible para este entorno. En cuanto a la distribución, Ubuntu es conocida por su estabilidad, su compatibilidad, su amplia comunidad de soporte y por ser gratuita. La versión 11.04 era la última disponible cuando se inició el proyecto que se va a realizar.

- **JDK 6 ([3])**

El *Java Development Kit* es necesario para usar la tecnología Java EE. En la página de Oracle puede obtenerse gratuitamente la última versión disponible, en este caso la 6 *release* 18.

- **IDE Netbeans 7 ([4])**

Tener un entorno de desarrollo facilita la tarea del programador. Hay muchos en el mercado, pero se ha optado por NetBeans por ser gratuito y tener excelente compatibilidad con el resto de software empleado en el proyecto. La versión 7 era la última disponible cuando se inició el proyecto.

- **Glassfish 3.1.1 ([5])**

Este servidor de aplicaciones gratuito viene integrado en el entorno de desarrollo de NetBeans, lo cual es una gran ventaja a la hora de desarrollar e ir probando la aplicación. La versión 3.1.1 era la última disponible en el comienzo del proyecto.

- **Valgrind 3.6.1 ([6])**

Esta suite gratuita de herramientas de depuración de código C se puede obtener desde los repositorios de Ubuntu o desde la propia web de Valgrind. Desde dichos repositorios se puede obtener la versión 3.6.1, aunque en la web ya está disponible

la versión 3.7. Se ha optado por la de los repositorios por ser más fácil de instalar y porque la última no aportaba nada nuevo al proyecto que aquí se describe.

1.5 Esquema de la memoria

Para el desarrollo de los subobjetivos mencionados anteriormente la memoria tiene la siguiente estructura:

- Bloque I: Introducción
  - Capítulo 1. Introducción y objetivos

    Aquí se presenta el proyecto que se va a describir y se definen los objetivos que pretende conseguir. Además se explican aspectos relacionados con el desarrollo de todo el proyecto.
- Bloque II: Estado del arte
  - Capítulo 2. Modulo empresarial para Java Path Finder

    En este capítulo se hace un estudio de un proyecto ya desarrollado. Es interesante para el proyecto que se va a realizar por compartir varios aspectos importantes. Ha servido para comprender más fácilmente cómo abordar el diseño del servidor con Java. Partiendo de este diseño ha sido más fácil conseguir mejoras en la arquitectura diseñada.
  - Capítulo 3. GCC

    En este capítulo se hace un breve estudio de la herramienta utilizada para compilar código C.
  - Capítulo 4. Valgrind

    En este capítulo se pretende conseguir el primero de los subobjetivos definidos estudiando la suite de herramientas de validación Valgrind y concluyendo por qué es una opción viable para el proyecto.
  - Capítulo 5. Java EE y Glassfish

    En este capítulo se pretende conseguir el segundo de los subobjetivos definidos estudiando la tecnología JavaEE y el servidor de aplicaciones Glassfish llegando a la conclusión de que son aconsejables para el proyecto.
- Bloque III: Diseño, implementación y pruebas de rendimiento
  - Capítulo 6. Diseño de la plataforma

    Parte del tercer subobjetivo definido se cumple en este capítulo. Se describen los requisitos de la plataforma y la forma en que se pretenden conseguir.

- Capítulo 7. Implementación

  En este capítulo se completa el tercer subobjetivo definido llevando a cabo la implementación del diseño conseguido en el Capítulo 6. Se describen los diferentes módulos del software desarrollado para la integración de JavaEE y Valgrind.

- Capítulo 8. Pruebas de rendimiento

  En este capítulo se describen y llevan a cabo las pruebas necesarias para cumplir el cuarto y último subobjetivo definido.

- Bloque IV: Conclusiones y futuras líneas de trabajo
  - Capítulo 9. Conclusiones y futuras líneas de trabajo

    En este último capítulo se describe la forma de conseguir los objetivos definidos, asícomo las conclusiones obtenidas al finalizar el proyecto y las futuras líneas de trabajo que podrían seguirse.

- Bloque V: Apéndices
  - Apéndice A: Presupuesto
  - Apéndice B: Instalación del software necesario
  - Apéndice C: Aspectos específicos de implementación
  - Apéndice D: Aplicaciones de prueba
  - Apéndice E: Referencias e hiperenlaces

Capítulo 2

Módulo empresarial para Java Path Finder

Este módulo empresarial para Java Path Finder se trata de un proyecto realizado por un alumno de la Universidad Carlos III de Madrid ([1]). Se podría considerar el equivalente para Java del proyecto que se describe en esta memoria, por eso es interesante realizar un breve estudio sobre él. Los objetivos son los mismos, salvo que el proyecto actual tiene que ofrecer la posibilidad de validar código C en vez de Java.

En este capítulo se van a describir su estructura y sus principales características para finalmente concluir qué puede aportar para la realización del proyecto que se va a realizar.

2.1 Java Path Finder

JPF es un software de validación formal para código Java. Básicamente es una máquina virtual de Java (JVM), que ejecuta un programa no sólo una vez, sino teóricamente en todos sus posibles caminos, buscando defectos o errores de ejecución como deadlocks (abrazos mortales) o excepciones no manejadas a lo largo de todos los caminos potenciales, además de ciertas propiedades que se quieran observar, añadiendo ciertas clases como entrada de la aplicación. Si JPF encuentra un fallo, indica la ejecución que ha llevado hasta el mismo. En esto se diferencia de otros validadores, pues éstos no indican los pasos que se han dado hasta llegar al fallo.

Está implementado en Java, por lo que no se ejecutará tan rápido como Java normal. Es una máquina virtual ejecutándose sobre otra.

Aunque hay semánticas de ejecución de códigos Java ya definidos, JPF tiene su propia semántica para acceder a las ejecuciones de aplicaciones. El conjunto de instrucciones de la máquina virtual está representado por un conjunto de clases que pueden ser reemplazadas o extendidas.

El conjunto de instrucciones por defecto hace uso de la capacidad de JPF llamada opciones de ejecución (execution choices). JPF identifica puntos en las aplicaciones desde donde la ejecución podrá proceder de manera diferente y después,

sistemáticamente, analiza todos ellos. Las opciones típicas son las diferentes secuencias programadas o valores aleatorios, pero JPF permite introducir nuestro propio tipo de opciones como entradas de usuario o eventos de la máquina de estados.

El número de caminos de una aplicación puede crecer exponencialmente y normalmente lo hará hasta llegar a ser inmanejable, lo que se llama problema de explosión de estados. Para enfrentar este problema, se utiliza el método de coincidencia de estados (State Matching). Cada vez que se encuentra un punto de elección comprueba si ya ha encontrado un estado similar, en cuyo caso, abandona dicho camino para volver a un estado anterior inexplorado y continúa desde ahí. Es decir, JPF puede restablecer estados del programa.

Lo que JPF no puede hacer es ejecutar código nativo. Esto no es porque no se pueda conseguir, sino porque realmente no tendría mucho sentido; por ejemplo, las llamadas al sistema para escribir un archivo no pueden ser fácilmente revertidas. Sin embargo, hay solución para esto y es configurable, son las clases native peers (pares nativos) y model (modelo).

Las clases native peers, contienen métodos que son ejecutados en vez de los métodos nativos reales. Este código es ejecutado por la máquina virtual de Java, no por la de JPF, lo cual, puede, además, acelerar el proceso. Las clases model son simplemente sustitutas de las clases estándar como java.lang.Thread, que ofrece alternativas para los métodos nativos que son mucho más manejables por JPF.

En resumen, JPF se puede considerar como un marco general para técnicas de verificación de ejecución de código Java, ofreciendo un rico conjunto de mecanismos de configuración y abstracción, además de ser muy extensible. Hay fallos en las aplicaciones que sólo JPF puede encontrar, por lo que se define como una herramienta muy valiosa para aplicaciones con una misión crítica donde el error no es una opción.

2.2 Diseño del módulo empresarial

El objetivo del proyecto es, básicamente, enviar prácticas a la aplicación para que sean validadas y devolver el resultado mediante correo. Por lo tanto, el diseño de esta aplicación web comprenderá la validación de prácticas vía página web, vía cliente de consola, el tratamiento de las prácticas enviadas (extracción y compilación), la posibilidad de gestión de archivos de validación y la notificación del resultado.

El proyecto tiene básicamente tres casos de uso: validación accediendo desde una página web, validación accediendo desde el cliente de consola y gestión de archivo de validación.

En los dos primeros casos de uso intervienen todos los módulos del sistema con el objetivo final de validar la aplicación, y enviar el resultado de nuevo al usuario o el administrador del sistema.

El tercer caso de uso únicamente concierne al administrador y permite al administrador poder subir sus propios archivos de validación para que, después al enviar la dirección web configurada al usuario, pueda incluirla. Este caso de uso incluye tres acciones:

- Subida de archivo.
- Eliminación de archivo.
- Vista del directorio.

2.3 Implementación del módulo empresarial

La implementación del módulo empresarial se ha realizado con la tecnología Java EE versión 5. Se ha estructurado en varios módulos que serán descritos a continuación:

Acceso a la aplicación

Se ha realizado mediante una opción de cliente web y otra, exclusiva del administrador, de cliente de consola. La primera está compuesta por una página JSP y la segunda por una clase Java. Ambas ofrecen las posibilidades de elegir modo Normal, Usuario o Experto, cada una de las cuales posee diferentes configuraciones.

Gestor de peticiones

Se ha realizado mediante un servlet de recepción que recoge los datos del usuario y seguidamente lo envía al siguiente módulo: gestor de accesos.

El gestor de accesos utiliza la tecnología JMS para implementar una cola de peticiones evitando así que el servidor se sature.

Extractor de archivos

Está implementado con dos clases Java que utilizan librerías externas para extraer archivos *.zip* y *.jar*.

Compilador

JPF utiliza los .class para ejecutarse, pero necesita los .java para referenciar los errores. Este módulo es el encargado de compilar los ficheros .java para obtener los .class. Además, gestiona la jerarquía de directorios empleados en cada petición.

Validador

Este módulo es el núcleo de la aplicación, pues se encarga de ejecutar JPF y obtener el resultado que el usuario espera. Está compuesto por dos clases Java. Una de ellas es un EJB sin estado.

Gestor de correos

Este módulo se encarga de enviar al usuario la respuesta obtenida en el módulo validador. Está implementado por una clase Java y hace uso de la librería Java para correo electrónico.

2.4 Resumen y conclusiones

El proyecto estudiado en este capítulo implementa una aplicación JEE que es capaz de recibir de un usuario, o del administrador, un programa Java, validarlo con la herramienta JPF y devolver el resultado mediante correo electrónico.

En principio se pensó en crear un módulo encargado de validar código C y ensamblarlo con el proyecto existente. Pero dicho proyecto no fue diseñado para ser ampliado de esta forma, y como resultado no se puede integrar un módulo extra con estas características. Es mucho menos costoso desarrollar un proyecto desde cero.

Sin embargo, hay ciertos aspectos que son útiles para la creación del nuevo proyecto. El diseño es reutilizable, en el sentido de que los objetivos son los mismos. Aunque se realicen algunos cambios, la filosofía es la misma. Además, aunque la implementación sea diferente por la naturaleza de las herramientas JPF y Valgrind, se pueden reutilizar módulos como el extractor y el gestor de correos.

Se opta finalmente por el desarrollo desde cero de una aplicación parecida a la ya existente, tomando referencias de ella, pero con las características necesarias para cumplir todos los objetivos definidos para el proyecto que se va a realizar.

Capítulo 3

GCC: GNU Compiler Collection

Para que el proyecto que se va a realizar funcione es necesario que compile código C. Por eso se ha optado por utilizar una herramienta de código libre que cumple con este requisito y que está muy extendida actualmente: GCC. Con ella, la aplicación va a ser capaz de compilar el código del usuario y crear un fichero ejecutable, que será usado posteriormente por Valgrind. En este capítulo se hace un breve estudio de esta herramienta.

3.1 Introducción

GNU Compiler Collection (colección de compiladores GNU) es un conjunto de compiladores creados por el proyecto GNU ([34], [35]). GCC es software libre y lo distribuye la FSF bajo la licencia GPL.

Estos compiladores se consideran estándar para los sistemas operativos derivados de UNIX, de código abierto o también de propietarios, como Mac OS X. GCC requiere el conjunto de aplicaciones conocido como binutils para realizar tareas como identificar archivos objeto u obtener su tamaño para copiarlos, traducirlos o crear listas, enlazarlos, o quitarles símbolos innecesarios.

Originalmente GCC significaba GNU C Compiler (compilador GNU para C), porque sólo compilaba el lenguaje C. Posteriormente se extendió para compilar C++, Fortran, Ada y otros.

3.2 Objetivo

GCC es parte del proyecto GNU, el cual tiene como objetivo mejorar el compilador usado en los sistemas GNU incluyendo la variante GNU/Linux. El desarrollo de GCC usa un entorno de desarrollo abierto y soporta muchas otras plataformas con el fin de fomentar el uso de un compilador-optimizador de clase global, para atraer muchos equipos de desarrollo, para asegurar que GCC y los sistemas GNU funcionen en diferentes arquitecturas y diferentes entornos, y más aún, para extender y mejorar las características de GCC.

3.3 Historia

Richard Stallman comenzó a escribir GCC en 1985. Él extendió un compilador existente para compilar C. El compilador originalmente compilaba Pastel, un dialecto extendido, no portable de Pascal, y estaba escrito en Pastel. Fue reescrito en C por Len Tower y Stallman, y publicado en 1987 como el compilador del Proyecto GNU, para tener un compilador disponible que fuera software libre. Su desarrollo fue supervisado por la Free Software Foundation (FSF).

En 1991, GCC 1.x había logrado un punto de estabilidad, pero limitaciones en la arquitectura prevenían muchas mejoras deseadas, así la FSF empezó a trabajar en GCC 2.x.

Como GCC era software libre, había programadores que querían trabajar en otras direcciones, por ejemplo, escribiendo interfaces para otros lenguajes distintos de C, éstos fueron libres de desarrollar sus propios forks del compilador. Múltiples bifurcaciones trajeron ineficiencia e inmanejabilidad, sin embargo, la dificultad de tener trabajo aceptado por el proyecto oficial de GCC era altamente frustrante para muchos. La FSF guardó un férreo control en qué era añadido a la versión oficial de GCC 2.x, esto hizo que GCC fuera usado como un ejemplo del modelo de desarrollo "catedral" en el ensayo de Eric S. Raymond titulado La catedral y el bazar.

Con la publicación de 4.4BSD en 1994, GCC llegó a ser el compilador por defecto de los sistemas BSD.

3.4 Lenguajes

En su versión 4.6 incluye front ends para C (gcc), C++ (g++), Java (gcj), Ada (GNAT), Objective-C (gobjc), Objective-C++ (gobjc++) y Fortran (gfortran).5 También esta disponible, aunque no de forma estándar, soporte para Go (gccgo), Modula-2, Modula-3, Pascal (gpc), PL/I, D (gdc), Mercury, y VHDL (ghdl).

3.5 Estructura

La interfaz exterior de GCC es generalmente estándar para un sistema UNIX. Los usuarios llaman un programa controlador llamado gcc, que interpreta los argumentos dados, decide que compilador usar para cada archivo y ejecuta el ensamblador con el código resultante, después posiblemente ejecuta el enlazador para producir un programa completo.

Cada uno de los compiladores es un programa independiente que toma como entrada código fuente y produce código en ensamblador. Todos ellos tienen una estructura interna común: un front end por lenguaje que procesa el lenguaje y produce

un árbol de sintaxis y un back end, que convierte esos árboles al lenguaje RTL (lenguaje de transferencia de registros) de GCC, luego realiza varias optimizaciones y produce el ensamblador utilizando un reconocimiento de patrones específico para la arquitectura, originalmente basado en un algoritmo de Jack Davidson y Chris Fraser.

Casi todo GCC está escrito en C, aunque gran parte del front end de Ada está escrito en Ada. El 30 de mayo de 2010 se anunció que se comenzará a utilizar C++ en el desarrollo de GCC.

Front ends

Los front ends varían internamente, teniendo que producir árboles que puedan ser manejados por el back end. Todos los analizadores son analizadores recursivos descendentes y fueron escritos manualmente, no generados automáticamente.

Hasta hace poco, el árbol de representación de programa no era totalmente independiente del procesador para el que se quería generar el código.

Recientemente se han incluido dos nuevas formas de árbol independientes del lenguaje. Estos nuevos formatos son llamados GENERIC y GIMPLE. El análisis ahora es realizado creando árboles temporales dependientes del lenguaje y convirtiéndolos a GENERIC. El gimplifier convierte esto a GIMPLE, que es el lenguaje común para un gran número de optimizaciones independientes de la arquitectura y del lenguaje.

La optimización en árboles no entra en lo que la mayoría de los desarrolladores de compiladores consideran trabajo del front end, ya que no es dependiente del lenguaje y no involucra el análisis. Los desarrolladores de GCC han dado a esta parte del compilador el nombre de middle end. Las optimizaciones incluyen eliminación de código que nunca se ejecuta, eliminación parcial de redundancia y redundancia a la hora de evaluar expresiones. Actualmente se está trabajando en optimizaciones basadas en dependencia de arreglos.

Back end

El comportamiento del backend está parcialmente especificado por el preprocesador de macros específicas a la arquitectura objetivo, por ejemplo para definir la posición de los bits más significativos, tamaño de palabra, convención para llamadas, etc. El backend utiliza éstas para la generación de RTL, aunque en GCC éste es independiente del procesador, la secuencia inicial de instrucciones abstractas es adaptada a la arquitectura objetivo.

3.6 Resumen y conclusiones

GCC es una herramienta con larga trayectoria capaz de compilar diferentes códigos de programación. Entre ellos se encuentra el código C, que es el se necesita compilar en el proyecto que se va a realizar.

Se va a incluir en la aplicación final por su fiabilidad y por su extenso uso en la mayoría de equipos Linux. Se encargará de compilar el código C en el módulo de compilación y de generar el ejecutable que utilizará Valgrind para la depuración.

Capítulo 4

Valgrind

El objetivo del proyecto es ofrecer al usuario la posibilidad de validar remotamente su código C. Esa validación va a realizarse mediante la suite Valgrind, por ser una de las más completas en este campo y adaptarse a las necesidades del proyecto. En este capítulo se va a hacer un estudio completo, para finalmente concluir por qué se utilizará en el desarrollo del proyecto que se va a realizar.

4.1 Introducción

Valgrind es una suite gratuita de código libre que ofrece una serie de herramientas de depuración que ayudan a hacer los programas en C más correctos y rápidos. Cada una de las herramientas se encarga de encontrar distintos tipos de problema relacionados, por ejemplo, con el uso de la memoria o la programación concurrente.

El precio a pagar es una notable pérdida de rendimiento; los programas se ejecutan entre cinco y veinte veces más lento al usar Valgrind, y su consumo de memoria es mucho mayor.

La arquitectura de Valgrind es modular, por lo que las herramientas incluidas en las nuevas versiones no alteran la estructura del software. Las herramientas actuales (versión 3.7) son las siguientes:

- **Memcheck**: permite detectar problemas de memoria, tales como fugas, uso de memoria no inicializada.
- **Cachegrind**: predice el uso de la memoria caché en los distintas ramas de ejecución del programa. Ayuda a que el programa sea más rápido.
- **Callgrind**: hace lo mismo que Cachegrind, pero aportando más información.
- **Helgrind**: es un detector de problemas en la programación con hilos, tales como las condiciones de carrera.
- **DRD**: igual que Helgrind, pero utiliza otro sistema, por lo que puede encontrar problemas diferentes.
- **Massif**: mide el rendimiento de la pila de memoria. Ayuda a que el programa utilice menos memoria.

- **DHAT**: es un tipo diferente de analizador de la pila de memoria. Ayuda a entender los problemas del uso de memoria reservada y evitar ineficiencias.
- **SGcheck**: es una herramienta experimental que permite detectar excesos en la pila de memoria y en *arrays* globales. Su funcionalidad es complementaria a la de Memcheck.
- **BBV**: herramienta experimental útil para la arquitectura de ordenadores.

De entre todas estas herramientas que Valgrind ofrece, se van a describir más adelante las dos más utilizadas: Memcheck y Helgrind. Son capaces de detectar los errores más comunes, y en general los que más diferencian un programa más correcto de otro que no lo es tanto.

4.2 Entendiendo Valgrind

En este apartado se describen los servicios que ofrece el núcleo de Valgrind, las opciones de la línea de comandos y el comportamiento en la ejecución ([25]).

4.2.1 Funcionamiento

Valgrind está diseñado para no ser intrusivo en la medida de lo posible. Trabaja directamente con los ficheros ejecutables y no es necesario recompilar, re-enlazar o hacer otro tipo de operaciones con el código fuente para que sea posible analizar el programa. El único requisito a la hora de compilar para que el resultado sea útil es hacerlo con la opción de depuración.

La herramienta por defecto es Memcheck, pero a la hora de ejecutar Valgrind se puede elegir cualquiera de las disponibles. Independientemente de cuál sea la elección, Valgrind toma el control del programa antes de empezar la ejecución. La información de depuración se toma del ejecutable y de las librerías asociadas, así los mensajes de error y los resultados de la validación pueden ser referenciados directamente sobre el código fuente, facilitando la tarea del usuario.

Después el programa es ejecutado en una CPU virtual que provee el núcleo de Valgrind. La herramienta seleccionada añade a la ejecución su propio código e instrumentación, para luego devolver al núcleo el resultado. La cantidad de "código extra" que se añade varía de unas herramientas a otras. Para hacerse una idea general, la herramienta que más código añade es Memcheck, y puede ralentizar la ejecución entre 10 y 50 veces. Por otro lado, la herramienta que menos código añade es Nulgrind, que no aporta ningún tipo de instrumentación extra, y aún así ralentiza la ejecución del programa 4 veces.

Valgrind simula cada instrucción que el programa analizado ejecuta. Por eso, la herramienta en uso comprueba no solo el código en la aplicación, sino todo el soporte que ofrecen dinámicamente las librerías enlazadas.

Valgrind es capaz de detectar errores que realmente pueden no interesar al usuario porque escapen de su ámbito de control, como son errores en el sistema operativo o en las librerías del sistema. Por eso, Valgrind permite filtrar ciertos tipos de errores. Estos filtros de supresión de errores son configurables por el usuario para ofrecer un servicio personalizado según sus intereses en cada momento.

4.2.2 El resultado

El resultado que nos devuelve Valgrind, independientemente de la herramienta que usemos, es una traza de texto. Cada línea tiene el formato que se ve en la figura 1.

```
=12345== some-message-from-Valgrind
```

Figura 1: Formato de línea del resultado de Valgrind (tomado de [25])

Se observa que entre la información que nos ofrece está el número de proceso, en el ejemplo es el proceso `12345`. Para evitar un exceso de información irrelevante, Valgrind filtra toda la información que puede ofrecer y solo devuelve en la traza de texto la más importante. El resto de información, que requeriría un complejo análisis para ser útil, queda guardada en ficheros del sistema. Valgrind pretende que la información que devuelve al usuario sea legible para él, facilitando la tarea de depuración que está llevando a cabo.

Valgrind ofrece tres opciones para devolver el resultado. Estas opciones son la salida estándar del sistema, un fichero de texto o un *socket* de red.

4.2.3 Los errores

Cuando se detecta un error en la ejecución del programa se escribe un mensaje en la traza de texto. En la figura 2 se puede ver un ejemplo.

```
==25832== Invalid read of size 4
==25832==    at 0x8048724: BandMatrix::ReSize(int, int, int) (bogon.cpp:45)
==25832==    by 0x80487AF: main (bogon.cpp:66)
==25832==  Address 0xBFFFF74C is not stack'd, malloc'd or free'd
```

Figura 2: Ejemplo de mensaje de error en el resultado de Valgrind (tomado de [25])

Este mensaje dice que el programa del ejemplo hizo una lectura ilegal en la dirección `0xBFFFF74C`, tal y como podría indicar Memcheck. La lectura está ocurriendo en la línea 45 de un fichero llamado `bogon.cpp`, llamada desde la línea 66 del mismo fichero. Para errores asociados con un bloque de memoria identificado, Valgrind reporta no solo la localización donde ocurrió el error, sino también donde estaba el bloque de memoria.

Valgrind recuerda todos los errores. Cuando se detecta un error, es comparado con los anteriores para ver si está duplicado. Si es así, el error es comunicado pero no se incluye un mensaje completo, así se evita la redundancia con duplicados.

En general los reportes de error se llevan a cabo antes de ejecutar la instrucción. Por ejemplo si se está usando Memcheck y el programa intenta leer de la dirección cero, Memcheck emitirá un mensaje a tal efecto, y el programa después probablemente morirá por una violación de segmento.

Es recomendable ir solucionando los errores del programa en el mismo orden en que aparecen en el resultado de Valgrind. De lo contrario, la tarea podría ser demasiado confusa.

El proceso de detectar los errores es bastante pesado y puede ralentizar la ejecución del programa, sobre todo si éste tiene demasiados errores. Para evitar serios problemas, Valgrind deja de recolectar errores cuando llega a 1.000 errores diferentes en el programa, o 10.000.000 errores en total (hay que tener en cuenta que no están incluidos en estos límites los errores filtrados). Llegado ese momento, habrá que empezar a solucionar errores, puesto que Valgrind no dirá nada más después de esos límites. Éstos son configurables, luego pueden aumentarse si hiciese falta.

4.2.4 Las opciones

Como se ha mencionado antes, Valgrind admite opciones a la hora de ser ejecutado. Las opciones están clasificadas de la siguiente manera:

- Selección de herramienta:

  `--tool=<toolname> [default: memcheck]`
  Se ejecutará la herramienta llamada `<toolname>`.

- Opciones básicas:

  Algunas de las opciones básicas se exponen en la tabla 1.

Opción	Función
`-h  --help`	Muestra ayuda para todas las opciones tanto del núcleo como de la herramienta seleccionada.
`--help-debug`	Igual que la anterior pero incluye información de depuración útil para desarrolladores.
`--version`	Muestra la versión del núcleo de Valgrind.
`-v  --verbose`	Muestra más información en el resultado.
`--trace-children=<yes/no>`	Valgrind informará también de los subprocesos en ejecución. Necesario en programación multihilos.
`--time-stamp=<yes/no>`	Cada línea en la traza del resultado irá precedida por una marca de tiempo.
`--log-file=<filename>`	Especifica el fichero de texto de salida.
`--log-socket=<ip-address:port-number>`	Especifica el socket de red de salida.

Tabla 1: Opciones básicas para ejecutar Valgrind

- Opciones relacionadas con los errores.
- Opciones relacionadas con la memoria.
- Opciones poco frecuentes.
- Opciones de depuración.
- Opciones de configuración.

4.3 Memcheck

Es la herramienta por defecto en Valgrind. Permite detectar los siguientes problemas de memoria:

- **Uso de memoria no inicializada.**

  Como se ve en el ejemplo de la figura 3, Memcheck advierte del error en la primera línea y en las siguientes detalla dónde ha sucedido.

```
Conditional jump or move depends on uninitialised value(s)
  at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
  by 0x402E8476: _IO_printf (printf.c:36)
  by 0x8048472: main (tests/manuel1.c:8)
```

Figura 3: Ejemplo de error de uso de memoria no inicializada

- **Lectura/escritura de memoria que ha sido previamente liberada.**

  En la figura 4 se ve un ejemplo de lectura inválida. Memcheck comunica el error en la primera línea y en las siguientes lo detalla.

```
Invalid read of size 4
  at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40B07FF4: read_png_image(QImageIO *) (kernel/qpngio.cpp:326)
  by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

Figura 4: Ejemplo de error de lectura ilegal

- **Lectura/escritura fuera de los límites de bloques de memoria dinámica.**

  En la figura 5 se puede ver un ejemplo del resultado que daría Memcheck para un error de este tipo.

```
Mismatched free() / delete / delete []
  at 0x40043249: free (vg_clientfuncs.c:171)
  by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
  by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
  by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
  at 0x4004318C: operator new[](unsigned int) (vg_clientfuncs.c:152)
  by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
  by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
  by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

Figura 5: Ejemplo de error en memoria dinámica

- **Fugas de memoria.**

  En la figura 6 se pueden ver dos ejemplos del resultado de Memcheck para errores de fugas de memoria.

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:39)

88 (8 direct, 80 indirect) bytes in 1 blocks are definitely lost in loss record 13 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:25)
```

Figura 6: Ejemplo de error de fuga de memoria

Problemas como estos son difíciles de detectar por otros medios. Pueden permanecer sin ser descubiertos durante cierto tiempo provocando situaciones impredecibles en tiempo de ejecución.

4.4 Helgrind

Es una herramienta de Valgrind para detectar errores de sincronización en C, C++ y Fortran que usan las primitivas de programación concurrente de POSIX ([26]).

Las abstracciones principales de POSIX *pthreads* son: un conjunto de hilos compartiendo un espacio de direcciones común, creación de hilos, arranque y parada de hilos, *mutex*, variables de condición, cerrojos de lector-escritor, *spinlocks*, semáforos y barreras.

Helgrind puede detectar estos tres tipos de errores:

- **Malos usos de la API de POSIX *pthreads*.**

  En la figura 7 se puede ver un ejemplo del mensaje de error de Helgrind en este caso.

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
   at 0x4C2408D: pthread_mutex_unlock (hg_intercepts.c:492)
   by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
   by 0x40079B: main (tc09_bad_unlock.c:50)
 Lock at 0x7FEFFFA90 was first observed
   at 0x4C25D01: pthread_mutex_init (hg_intercepts.c:326)
   by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
   by 0x40079B: main (tc09_bad_unlock.c:50)
```

Figura 7: Ejemplo de error por mal uso de POSIX pthreads

- **Bloqueos potenciales derivados de la petición de cerrojos.**

En la figura 8 se encuentra el mensaje de error que ofrece Helgrind para un ejemplo de mala sincronización.

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF0006A0" violated

Observed (incorrect) order is: acquisition of lock at 0x7FF0006A0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x400825: main (tc13_laog1.c:23)

 followed by a later acquisition of lock at 0x7FF0006D0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x400853: main (tc13_laog1.c:24)

Required order was established by acquisition of lock at 0x7FF0006D0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x40076D: main (tc13_laog1.c:17)

 followed by a later acquisition of lock at 0x7FF0006A0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x40079B: main (tc13_laog1.c:18)
```

Figura 8: Ejemplo de error de mala sincronización

- **Condiciones de carrera: acceder a partes concretas de memoria sin la adecuada sincronización o reserva de acceso.**

  En la figura 9 se aprecia el tipo de mensaje de error que ofrece Helgrind cuando encuentra una condición de carrera.

```
Thread #1 is the program's root thread

Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
   by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
   by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
   by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
   at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x4005DC: child_fn (simple_race.c:6)
   by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
   by 0x511C0CC: clone (in /lib64/libc-2.8.so)

Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```

Figura 9: Ejemplo de error de condición de carrera

Problemas como estos son difíciles de detectar, puesto que no siempre ocurren y dependen de un factor temporal que no se puede controlar.

Helgrind trabaja mejor cuando el programa utiliza solo la API de POSIX *pthreads*. Sin embargo, se pueden usar primitivas de programación multihilos personalizadas describiendo su comportamiento a Helgrind mediante las macros `ANNOTATE_*` definidas en el fichero `helgrind.h`.

4.5 Otras opciones

Otra opción que ha sido valorada para la realización del proyecto es MyGCC ([9]). También es de código abierto y ofrece unos servicios similares a Valgrind. Sin embargo, es menos completa que la suite finalmente utilizada y su desarrollo se encuentra parado desde hace unos años. Por el contrario, Valgrind tiene una importante comunidad de desarrolladores detrás y se pueden encontrar nuevas versiones del software cada cierto tiempo.

4.6 Resumen y conclusiones

Valgrind es una suite gratuita y de código libre que ofrece diversas herramientas útiles para la depuración de código C. Entre sus aplicaciones se encuentran la de detectar problemas de memoria y de concurrencia, problemas que no deben existir en una aplicación considerada correcta. Valgrind ofrece una amplia gama de herramientas y opciones que facilitarán al programador la tarea de depuración permitiéndole establecer preferencias a la hora de ejecutar las herramientas.

Llegado este punto, se tienen conocimientos suficientes para concluir que Valgrind es un software adecuado para el propósito del proyecto, ya que cumple con los requisitos establecidos de proveer un servicio de detección de problemas de memoria y de programación concurrente. Además, del estudio se obtiene la información necesaria para saber cómo utilizar la suite.

Otra ventaja es que Valgrind está en constante desarrollo y evolución, por lo que futuras versiones pueden ser incluidas en el proyecto para ofrecer mejor servicio sin que esto suponga ningún cambio drástico en la plataforma.

Capítulo 5

JEE6 y Glassfish

Para conseguir el objetivo final se necesita desarrollar una plataforma capaz de ofrecer al usuario una interfaz para que aporte sus datos y su programa C, y una infraestructura que procese, incluyendo la parte de validación, toda esa información.

Para tal efecto se encuentra la tecnología Java EE, y Glassfish dentro de ella. En este capítulo se analizan ambas tecnologías. Se profundizará en las características que puedan ser interesantes y permitan el acceso de un gran número de usuarios.

5.1 Introducción a Java EE

Java EE ([10]) es una plataforma de programación que permite desarrollar y ejecutar software de aplicaciones en Java con una arquitectura distribuida de n niveles, basándose en componentes software modulares ejecutados sobre un servidor de aplicaciones. Su estructura está representada en la figura 10.

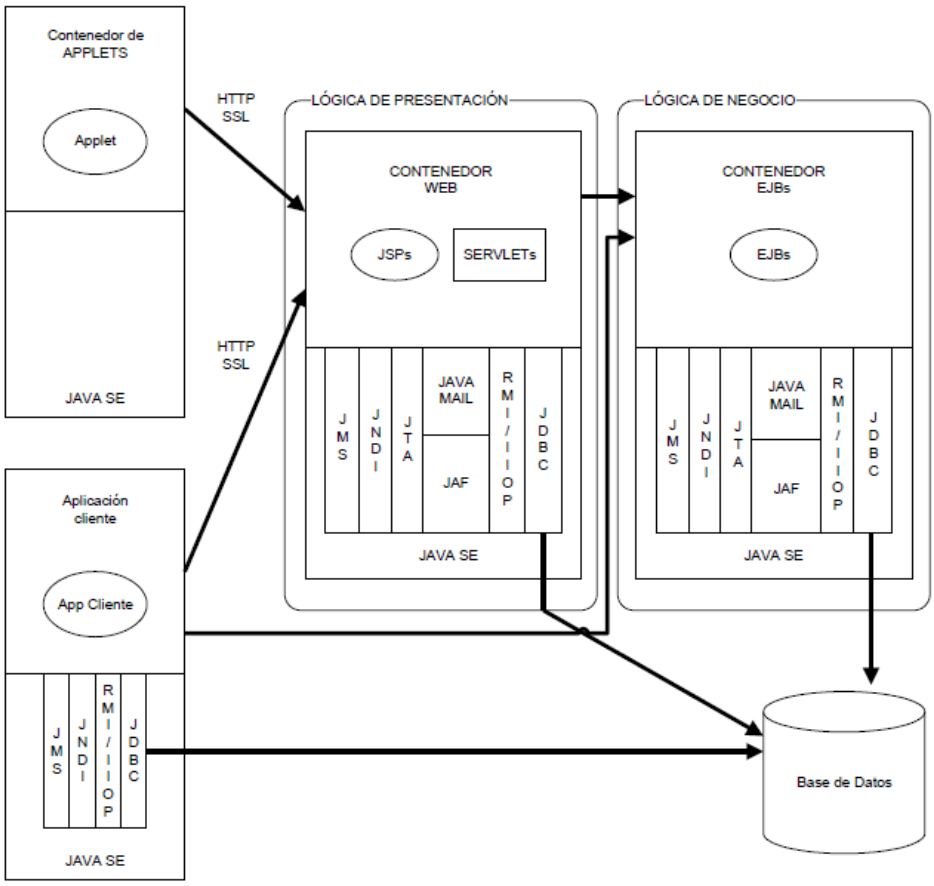

Figura 10: Estructura de una aplicación Java EE (tomada de [1])

Todo esto permitirá crear una aplicación empresarial portable entre plataformas, escalable e integrable con las tecnologías anteriormente mencionadas, lo cual hace que ésta sea una opción muy interesante para el proyecto. Otros beneficios añadidos son que el servidor de aplicaciones puede manejar transacciones, seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, de forma que para el desarrollador la tarea se reduce al diseño de la lógica de negocio.

5.1.1 APIs

En esta sección se describen brevemente algunos de los diferentes paquetes y APIs que contiene la tecnología Java EE. Son los siguientes:

- **APIs Generales:** Son APIs que extienden la funcionalidad de las APIs base de Java SE.
- **EJBs (javax.ejb):** La API Enterprise JavaBeans (EJBs) define un conjunto de APIs que un conjunto de objetos distribuidos soportará para suministrar persistencia, RPCs (llamadas remotas RMI o RMI-IIOP), control de concurrencia, transacciones y control de acceso para objetos distribuidos.

- **JNDI (javax.naming):** Los paquetes javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldap y javax.naming.spi definen la API de Java Naming and Directory Interface o JNDI. Permite a los clientes descubrir y buscar objetos y nombres a través de un nombre y será independiente de la aplicación subyacente.
- **JDBC (java.sql):** Los paquetes java.sql y javax.sql definen el API de JDBC. JDBC es un API que maneja la conectividad Java con base de datos. Define métodos para peticiones y actualizaciones de datos de la base de datos.
- **JTA (java.transaction):** Estos paquetes definen la Java Transaction API (JTA). JTA establece una serie de interfaces Java entre el manejador de transacciones y las partes involucradas en el sistema de transacciones distribuidas: el servidor de aplicaciones, el manejador de recursos y las aplicaciones transaccionales.
- J**AXP (javax.xml):** Estos paquetes definen el API de Java API for XML Processing (JAXP). Es un API que se encarga de la manipulación y tratamiento de archivos XML.
- **JMS (javax.jms):** Estos paquetes definen el API de Java Messaging Service (JMS). JMS es un estándar de mensajería de Java EE que permite crear, enviar, recibir y leer mensajes. Provee un servicio fiable y flexible para un intercambio asíncrono de información de negocio crítica.
- **JPA (javax.persistence):** Este paquete provee las clases e interfaces para gestionar la interacción entre los proveedores de persistencia, las clases administradas y los clientes del API de persistencia Java (JPA). JPA busca unificar la manera en que funcionan las utilidades que proveen un mapeo objeto-relacional. Su utilidad es no perder las ventajas de orientación a objetos al interactuar con la base de datos y permitir utilizar objetos regulares (POJOs – Plain Old Java Objects).

5.2 EJBs: Enterprise JavaBeans

La tecnología EJB ([11], [12]) es la arquitectura de componentes del lado del servidor de la plataforma Java EE. Permite un desarrollo rápido y simplificado de aplicaciones distribuidas, transaccionales, seguras y portables para la tecnología Java. Forma parte de la lógica de negocio de una aplicación empresarial Java.

5.2.1 Versiones de los EJBs

La especificación EJB ha ido evolucionando a la par que lo hacía la propia especificación J2EE. Las diferentes versiones que han existido hasta la fecha son:

- **EJB 1.0:** la especificación original.
- **EJB 1.1:** la primera incluida dentro de J2EE.

- **EJB 2.0:** incluida en J2EE 1.3 añadía las interfaces Locales y los Message-Driven beans.
- **EJB 2.1:** incluida en la última revisión de J2EE, la 1.4.
- **EJB 3.0:** Ahora con Cluster y está incluida en JEE 5.1 (La última estable).
- **EJB 3.1:** incluida en JEE 6 en diciembre de 2009.

5.2.2 Java EE 6 y EJB 3.0: Versiones para el proyecto

EJBs 3.0 es un API que forma parte del estándar de construcción de aplicaciones empresariales Java EE 6 ([14]). Existen tres tipos diferentes de EJBs:

- **EJB de Entidad (*Entity Bean*):**

Encapsulan los objetos del lado del servidor que almacena los datos. Presentan la característica fundamental de persistencia. En EJB 3.0 los Entity Beans pasan a pertenecer al API JPA y cambia la forma de implementación.

JPA es el API de persistencia desarrollada para Java EE e incluida en el estándar EJB 3. Su objetivo es no perder las ventajas de la orientación a objetos al interactuar con una base de datos, y permitir el uso de POJOS. Consta de:

- *Java Persistence API*
- *Query Language*
- *Object relational mapping metadata*

Ciclo de vida: El control de un Entity Bean reside en el API EntityManager. Cuando se produce un evento en dicho Bean, EntityManager llama a los métodos lifecycle-callbacks si la clase los ha implementado. Éstos son: PrePersist, PostPersist, PreRemove, PostRemove, PreUpdate, PostUpdate, PostLoad.

- **EJB de sesión (*Session Bean*):**

Gestionan el flujo de información del servidor. Representan procesos ejecutados en respuesta a una solicitud del cliente. Puede haber de dos tipos:

  - Con estado (stateful): Los beans de sesión con estado son objetos distribuidos que poseen un estado, éste estado no es persistente, pero el acceso al bean se limita a un solo cliente. Su ciclo de vida se puede ver en la figura 11.

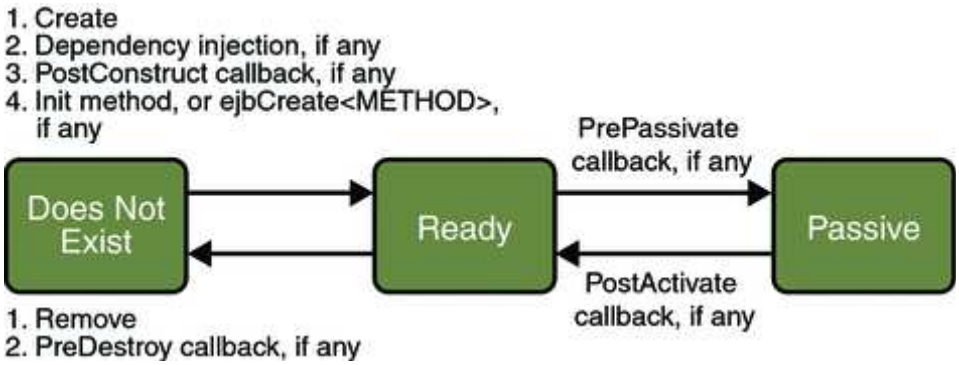

Figura 11: Ciclo de vida de un Stateful Session Bean (tomada de [14])

- Sin estado (stateless): Son objetos distribuidos que carecen de estado asociado, permitiendo por tanto que se los acceda concurrentemente. No se garantiza que los contenidos de las variables de instancia se conserven entre llamadas al método. Su ciclo de vida se puede ver en la figura 12.

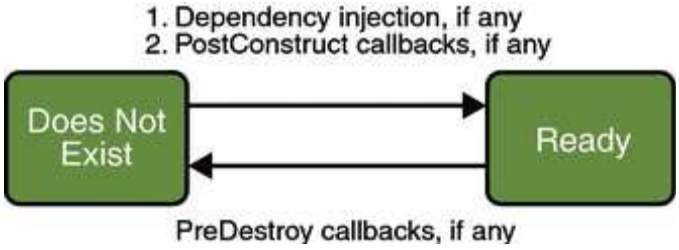

Figura 12: Ciclo de vida de un Stateless Session Bean (tomada de [14])

- **EJB de mensajes (*Message-driven Bean*)**

Son los únicos beans con funcionamiento asíncrono. Utilizan el Java Messaging System (JMS), se suscriben a un tema (topic) o a una cola (queue) y se activan al recibir un mensaje dirigido a dicho tema o cola. No requieren de instanciación por parte del cliente. Su ciclo de vida se puede ver en la figura 13.

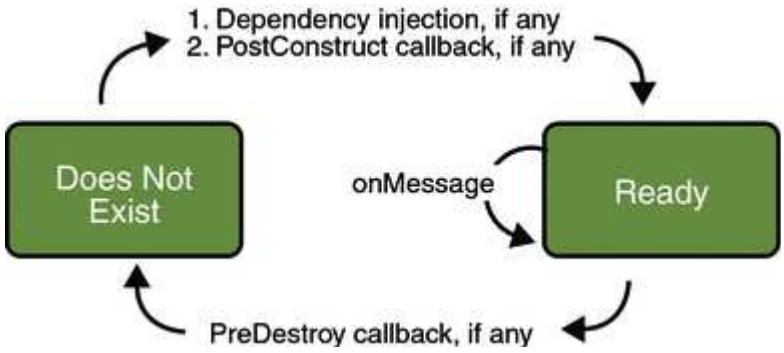

Figura 13: Ciclo de vida de un Message Bean (tomada de [14])

5.2.2.1 Interfaces Local y Remota de EJBs 3.0

Interfaz Remota: Los Beans ofrecen métodos concretos que pueden ser llamados por un cliente, aunque pueden existir métodos que no sean públicos. Para ello, a cada Bean le corresponde una interfaz ya sea Local o Remota, como mínimo un Bean deberá implementar una de ellas. La interfaz Remota es para aquellos Beans que puedan ser llamados desde una máquina virtual de Java distinta a donde se encuentra el servidor de aplicaciones. Incluye la anotación @Remote.

Interfaz Local: Normalmente se utilizan para ofrecer servicios a otros Session Beans que se encuentran en el mismo servidor de aplicaciones, es decir, en la misma máquina virtual, donde pueden tratarse de métodos que aparezcan en un Bean cuya interfaz es remota. Ya no es necesario incluir métodos del ciclo de vida en ésta interfaz, como se podía hacer en la versión 2.1. Incluye la anotación @Local.

5.2.2.2 Cambios introducidos por la versión EJB 3.0

El modelo de programación propuesto por la versión 2.1 conllevaba una serie de inconvenientes que limitaron mucho el uso de esta especificación. El objetivo de los EJBs 3.0 es simplificar el desarrollo de aplicaciones Java y estandarizar el API de persistencia para la plataforma Java. Las principales características son:

1. **Metadata Annotation (Anotaciones Java):** EJB 3.0 utiliza anotaciones para simplificar el desarrollo de componentes. Ahora no hay necesidad de escribir un descriptor de despliegue, pero todavía lo soporta. Las anotaciones podrán ser sobrescritas por dicho descriptor.

2. **Encapsulamiento de dependencias del entorno:** Ahora se utilizan las anotaciones y el mecanismo de inyección de dependencias para encapsular las dependencias del entorno y el acceso a JNDI.
3. **Especificación de EJBs más simplificada:** Ahora ya no hay necesidad de escribir interfaces locales y de componente ni de implementar la interfaz javax.ejb.EnterpriseBean para la clase EJB. La clase EJB es ahora una clase pura de Java, también conocida como POJO, y la interfaz es conocida como POJI, una simple interfaz Java. Por esto se podrá desarrollar una aplicación empresarial mucho más rápido.
4. **Inyección de dependencia:** El API de búsqueda y la utilización del entorno y las referencias a recursos de los EJBs se ha simplificado utilizando anotaciones.
5. **Simplificación de la persistencia:** La persistencia de los objetos entidad, ahora es más simple a través de la introducción del API de persistencia de Java (JPA). Se introduce un nuevo API llamado EntityManager, el cual se utiliza para crear, encontrar, eliminar y actualizar entidades. Ahora estos objetos soportarán herencia y polimorfismo. Para establecer dichas relaciones en EJB 3.0 se utilizarán anotaciones.

6. **Interfaces de Callback:** Elimina la necesidad de implementar métodos de callback de ciclo de vida innecesarios.
7. **Timer Service:** Se suele utilizar cuando se desea realizar un proceso controlado temporalmente, ya sea en una fecha determinada o cada cierto tiempo. Se suele utilizar en Stateless Session Beans y Message Driven Beans (MDBs).
8. **Interceptors y Entity Listeners:** Un interceptor es una clase cuyos métodos se ejecutan cuando se llama al método de otra clase totalmente diferente. Se podrá configurar para Session Beans y MDBs, pero no para los Entities.
9. **Web Services:** Cualquier servicio que pueda llamarse mediante los protocolos utilizados en el World Wide web por un cliente situado remotamente. Toda la comunicación está basada en XML (petición y respuesta). Estos XMLs se crean bajo el estándar SOAP (Simple Object Access Protocol). SOAP define reglas de serialización, empaquetado de datos y generación de mensajes. En EJB 3.0 se pueden utilizar anotaciones para su implementación.

5.3 Servlets y Java Server Pages

Estas tecnologías del lado del servidor se incluyen en el estándar Java EE. Forman parte de la lógica de presentación de una aplicación empresarial Java.

5.3.1 Servlets

Un servlet es un objeto que se ejecuta en un servidor o en un contenedor Java EE, especialmente diseñado para ofrecer contenido dinámico desde un servidor web, generalmente HTML. Otra opción que permite generar contenido dinámico son los JSPs (Java Server Pages).

Un servlet implementa la interfaz javax.servlet.Servlet o hereda de alguna de las clases convenientes para un protocolo específico, por ejemplo, javax.servlet.HttpServlet. Al implementar esta interfaz, el servlet es capaz de interpretar los objetos del tipo HttpServletRequest y HttpServletResponse, quienes contienen la información de la página que invocó el servlet. Las invocaciones que admite son de tipo GET, POST o PUT entre otras y cada una de ellas tendrá un método específico en el servlet que las controlará.

Entre el servidor de aplicaciones y el servlet existe un contrato que determina cómo han de interactuar.

5.3.1.1 Ciclo de vida

El ciclo de vida de un servlet se divide en los siguientes puntos:

1. El cliente solicita una petición al servidor vía URL
2. El servidor recibe la petición:
    - Si es la primera, se utiliza el motor de servlets para cargarlo y se llama al método init().
    - Si ya está iniciado, cualquier petición se convierte en un nuevo hilo. Un servlet puede manejar múltiples peticiones de clientes.
3. Se llama al método service() para procesar la petición devolviendo el resultado al cliente.
4. Cuando se apaga el motor de un servlet se llama al método destroy() que lo destruye y libera los recursos.

5.3.2 Java Server Pages (JSPs)

JSPs es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Las JSPs permiten la utilización de código Java mediante scripts. Además, es posible utilizar algunas acciones JSP predefinidas mediante etiquetas. Estas etiquetas pueden ser enriquecidas mediante la utilización de bibliotecas de etiquetas (TagLibs o Tag Libraries) externas e incluso personalizadas. Se puede considerar una alternativa a los servlets o se pueden utilizar de forma complementaria.

Los JSPs son en realidad servlets: un JSP se compila a un programa en Java la primera vez que se invoca y del programa en Java se crea una clase que se empieza a ejecutar en el servidor como un servlet. La principal diferencia entre los servlets y los JSPs es el enfoque de la programación: un JSP es una página web con etiquetas especiales y código Java incrustado, mientras que un servlet es un programa Java puro que recibe peticiones y genera a partir de ellas una página web.

5.4 Servidor de aplicaciones

Para poder ofrecer la herramienta Valgrind vía web, debe estar integrada con la tecnología Java EE y estar montado todo en un servidor de aplicaciones, para que la aplicación final Java EE funcione. En este capítulo se analizan los servidores de aplicaciones y en especial el servidor Glassfish de Sun ([5]), por ser de código libre y por su sencillez a la hora de su instalación y configuración.

5.4.1 Definición

El concepto de servidor de aplicaciones está relacionado con el concepto de sistema distribuido. Un sistema distribuido, en oposición a un sistema monolítico, permite mejorar tres aspectos fundamentales en una aplicación:

- Alta disponibilidad.
- Escalabilidad.
- Mantenimiento.

El estándar Java EE permite el desarrollo de aplicaciones empresariales de forma sencilla y eficiente. Una aplicación desarrollada con dicha tecnología podrá ser desplegada en cualquier servidor de aplicaciones que cumpla con el estándar. Un servidor de aplicaciones por tanto es una implementación de la especificación Java EE que proporcionará servicios que soportan la ejecución y disponibilidad de las aplicaciones desplegadas. Es el corazón de un gran sistema distribuido y proporciona una estructura en tres capas que permite estructurar nuestro sistema de forma más eficiente tal y como se puede ver en la figura 14.

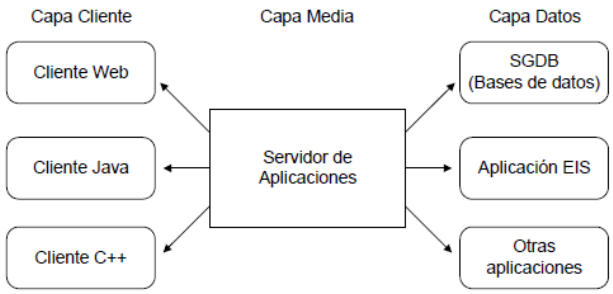

Figura 14: Capas de una aplicación empresarial (tomada de [1])

Los servidores de aplicaciones funcionan como contenedores de componentes de las aplicaciones JEE. Estos componentes son servlets, JSPs, y EJBs y permiten implementar diferentes capas de la aplicación, como la interfaz de usuario, la lógica de negocio, la gestión de sesiones de usuario o el acceso a bases de datos remotas.

Típicamente incluyen también middleware (o software de conectividad) que les permite intercomunicarse con diferentes servicios. Brindan además soporte a una gran variedad de estándares, como HTML, XML, IIOP, JDBC y SSL, que permiten su funcionamiento en ambientes web y la conexión a una gran variedad de fuentes de datos.

5.4.2 Diferentes servidores de aplicaciones

- BEA WebLogic ([17])
- JBoss ([18])
- IBM WebSphere ([19])
- Sun Glassfish ([5])
- Borland AppServer ([20])

De éstos servidores los más populares en la actualidad, por ser libres, son JBoss y Glassfish de Sun. En el proyecto se utilizará Glassfish entre otras cosas por su sencillez en la instalación, configuración y por ser de código abierto.

5.4.3 Servidor Glassfish de Sun

Glassfish de Sun es un servidor de aplicaciones sencillo y rápido basado en la plataforma Java EE para el desarrollo de aplicaciones empresariales. En la tabla 2 se pueden ver sus características y funcionalidades.

Características	**Beneficios**
Compatible con Java EE 5 y 6	Implementa las últimas versiones de Java EE y está continuamente actualizándose.
Mejora la productividad del desarrollador	Con APIs Java EE simplificadas y anotaciones, se reduce el tamaño del código a desarrollar, por lo que se mejora en la productividad.
Arquitectura orientada a servicios (SOA)	Soporta JAX-WS 2.0, JAXB 2.0, y Open ESB, proveyendo una arquitectura abierta y extensa para la colaboración entre tecnologías de integración y servicios web.
Herramientas de desarrollo	Puede ser integrado con IDEs como NetBeans y Eclipse para un desarrollo más cómodo y eficiente.
Interfaz web sencilla	El servidor posee una interfaz web muy intuitiva que permite configurar completamente el funcionamiento del servidor de forma sencilla.
Instalación sencilla	Su instalación no conlleva mucha dificultad y existen multitud de guías para este fin.

Documentación abundante	Hay mucha documentación sobre uso, administración y desarrollo, lo cual es muy útil.
Código libre	Se puede obtener gratuitamente.

Tabla 2: Características del servidor de aplicaciones Glassfish (tomada de [1])

5.5 Resumen y conclusiones

JavaEE, es una plataforma de programación para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java con arquitectura de N capas distribuidas y que se apoya ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones. La plataforma Java EE está definida por una especificación. Similar a otras especificaciones del Java Community Process, Java EE es también considerada informalmente como un estándar debido a que los proveedores deben cumplir ciertos requisitos de conformidad para declarar que sus productos son conformes a Java EE

Java EE incluye varias especificaciones de API, tales como JDBC, RMI, e-mail, JMS, Servicios Web, XML, etc y define cómo coordinarlos. Java EE también configura algunas especificaciones únicas para Java EE para componentes. Estas incluyen Enterprise JavaBeans, servlets, portlets (siguiendo la especificación de Portlets Java), JavaServer Pages y varias tecnologías de servicios web. Ello permite al desarrollador crear una Aplicación de Empresa portable entre plataformas y escalable, a la vez que integrable con tecnologías anteriores. Otros beneficios añadidos son, por ejemplo, que el servidor de aplicaciones puede manejar transacciones, la seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, significando que los desarrolladores pueden concentrarse más en la lógica de negocio de los componentes en lugar de en tareas de mantenimiento de bajo nivel.

Habiendo analizado la tecnología Java EE se ve que ofrece características muy útiles para el objetivo del proyecto que se va a realizar, como por ejemplo la tecnología JMS (Java Messaging Service) la cual permitiría controlar una cola de peticiones e ir atendiendo dichas peticiones de la forma que mas convenga mediante los EJBs de mensajes o MDBs (Message Driven Beans).

Además ofrece mecanismos para manejar las peticiones inicialmente, es decir, cuando se accede al servidor, como son los servlets y las JSPs. Éstos permiten crear una interfaz con la que los usuarios podrán interactuar, ya que generan código HTML.

Como se observa en la figura 10, hay cinco grandes módulos en la estructura. El contenedor de applets y la base de datos no se van a utilizar en el proyecto que se va a realizar, pues no se necesitan para obtener el resultado requerido en los objetivos. Sin embargo, los otros tres módulos si serán útiles.

El módulo de la aplicación cliente será el encargado de ofrecer al usuario la interfaz definida en el módulo de la lógica de presentación, el cual procesará los datos del usuario. Finalmente, el módulo de la lógica de negocio hará el resto de procesado.

Capítulo 6

Diseño de la plataforma

Ahora que se han analizado las tecnologías a emplear en el proyecto, se diseña la aplicación explicando el uso de la misma, los componentes que la forman y la interacción entre ellos.

La finalidad del proyecto es que un alumno pueda enviar sus prácticas de programación en C a la aplicación y que obtenga el resultado de la validación mediante un correo electrónico. Por lo tanto, el diseño de la aplicación web incluirá la recepción de datos del alumno, el almacenamiento de los mismos, extracción de los ficheros, compilación del programa, validación utilizando Valgrind y notificación de resultado.

Atendiendo a lo enterior, el caso de uso es el que se expone en la figura 15.

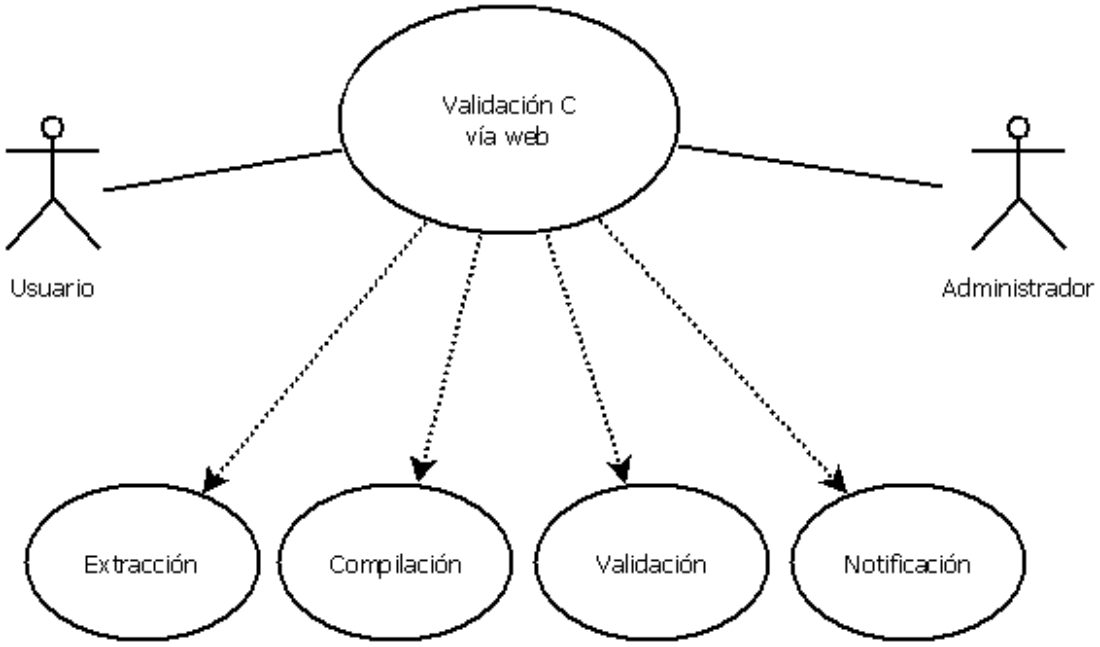

Figura 15: Caso de uso de la plataforma

6.1 Vista general

Para poseer una visión general del diseño de la plataforma completa hay que observar la figura 16. En ella se distinguen los principales bloques de los que va a constar el proyecto.

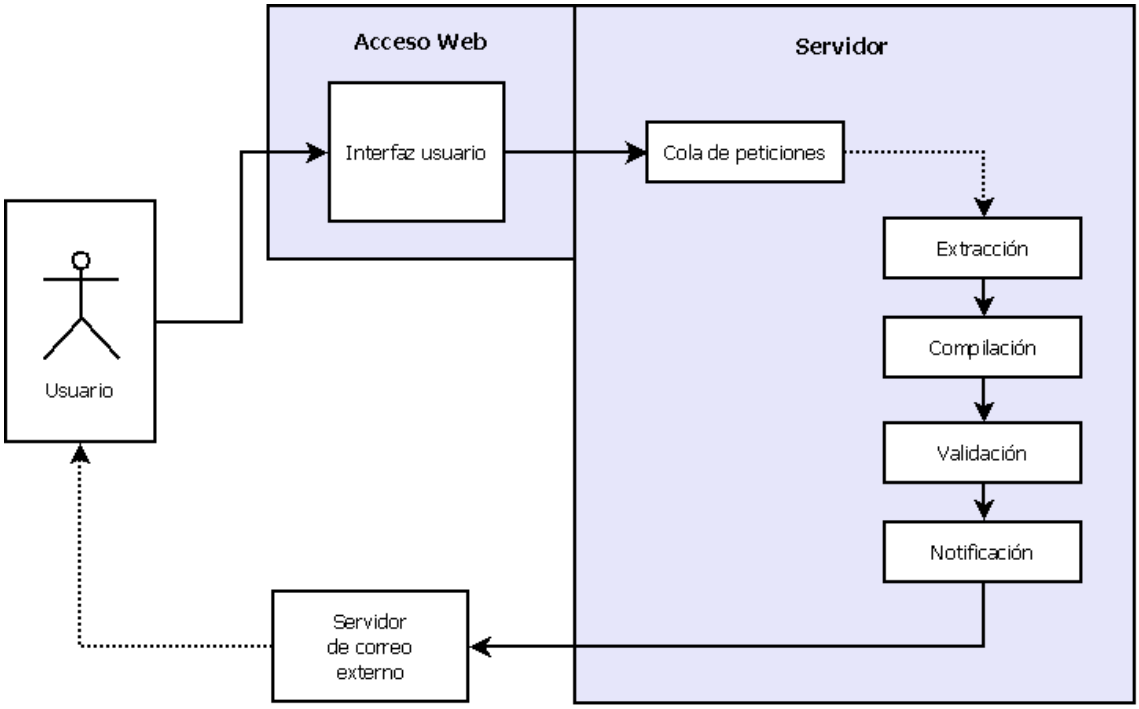

Figura 16: Diagrama de bloques de la plataforma

El primer bloque es el del usuario. Puede tratarse tanto de un usuario cualquiera como del propio administrador, siempre que se posea un navegador web. Será el administrador el que facilite la url para acceder al servicio.

El segundo bloque es el de acceso a la plataforma. En él el usuario tendrá la oportunidad de introducir sus datos y elegir el programa C de su equipo que desea validar.

Después se entra en el bloque del servidor, donde está toda la lógica de negocio. En él se encuentra la cola de peticiones, donde éstas esperarán a ser atendidas, para después pasar por el proceso de extracción, compilación, validación y notificación.

Finalmente se encuentra el bloque del servidor de correo externo. Éste es ajeno a la plataforma desarrollada aquí, así que el administrado deberá ofrecer uno con la correspondiente configuración.

6.2 Componentes de la aplicación

A continuación serán explicados brevemente los distintos componentes de la aplicación. Todos ellos son independientes, por lo que sería relativamente fácil actualizarlos o modificarlos siempre y cuando se respetaran las interacciones entre ellos (apartado 6.4).

- **Acceso Web:**

  Este módulo tiene dos subcomponentes. El primero será el encargado de proporcionar al usuario la posibilidad de introducir la información necesaria de forma visual. Dicha información incluye los datos identificativos del usuario, el código fuente de su programa comprimido en un fichero y las opciones relacionadas con la compilación y el modo de ejecución de Valgrind.

  El segundo subcomponente se encargará de comprobar que la información sea coherente y de almacenarla en un directorio específico. Después enviará la petición al gestor de peticiones.

- **Gestor de peticiones**

  Este módulo recibirá las peticiones de los usuarios y las encolará hasta que el servidor pueda procesarlas. De esta manera se evita que, si el servidor está saturado, el usuario reciba una negativa de servicio.

  Cuando el servidor pueda procesar otra petición, el gestor de peticiones enviará los datos de la petición correspondiente al siguiente módulo.

- **Extractor de archivos**

  Este módulo se encargará de extraer el código fuente comprimido en el fichero que el usuario ha enviado. Dicho código será almacenado en un directorio específico.

- **Compilador**

  Este módulo compilará el código fuente del usuario con las opciones que éste haya incluido. Si la compilación es exitosa, se guardará un ejecutable en un directorio específico. Si no, se informará al usuario del error para facilitarle la corrección.

- **Ejecutor**

  Este módulo ejecutará la herramienta de la suite Valgrind que el usuario haya elegido sobre el código fuente ya compilado. El resultado tendrá que ser enviado al usuario, pues esa es la finalidad del proyecto.

  Se impondrá un límite de tiempo para que programas con bucles infinitos no saturen el servidor. Si el programa en cuestión supera dicho límite, se informará al usuario para que actúe consecuentemente.

- **Gestor de correos**

  Este módulo será el encargado de enviar el correo electrónico con la información pertinente. Podrá incluir el resultado de la validación o el informe de error de alguno de estos módulos: extracción, compilación o ejecución.

6.4 Interacción entre componentes

En la figura 17 se puede ver la interacción entre los módulos descritos anteriormente, es decir, la información y/o recursos que utilizan para su función y la información y/o recursos que devuelven para que las utilice el módulo siguiente.

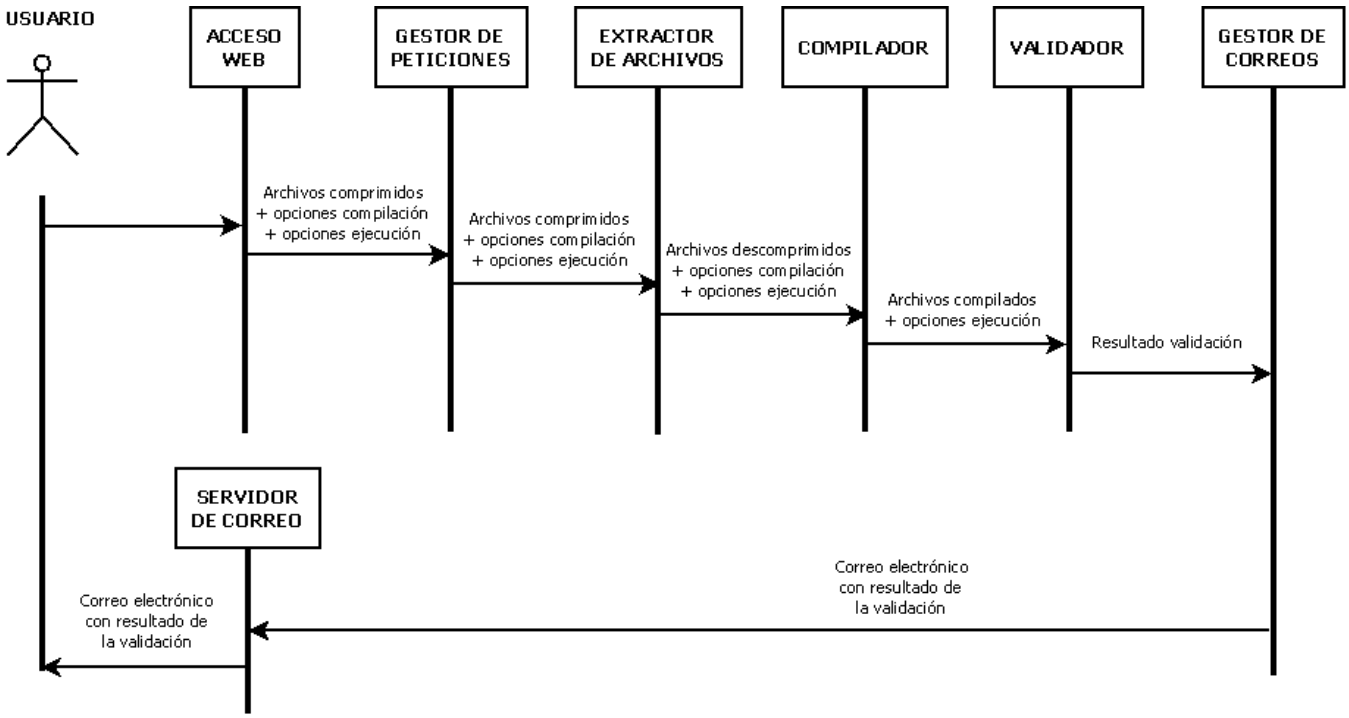

Figura 17: Diagrama de interacción entre componentes

Para empezar, el usuario accede a la plataforma a través de una url facilitada por el administrador. Una vez que está en el módulo de acceso web puede empezar a usar el servicio. Debe introducir sus datos, el fichero comprimido con el código a validar y unas opciones de compilación y ejecución.

Luego es el gestor de peticiones el encargado de almacenar la petición hasta que pueda ser atendida. Una vez llegado ese momento, pasa la información al extractor.

El módulo extractor se encarga exclusivamente de descomprimir el fichero y obtener el código fuente del programa del usuario. Ahora dicho código estará disponible para los siguientes módulos.

El módulo compilador utiliza el código fuente descomprimido y las opciones de compilación para crear un fichero ejecutable del programa compilado.

El módulo de ejecución utiliza tanto el fichero compilado anteriormente como las opciones de ejecución introducidas al principio. Es el núcleo de toda la aplicación, puesto que su resultado es el que interesa al usuario.

Dicho resultado es utilizado por el módulo de gestor de correos para crear un correo electrónico con ese contenido y enviarlo a la dirección del usuario mediante el servidor de correos externo.

El servidor de correo externo, que ya no depende del proyecto que se va a realizar, será el encargado de hacerle llegar al usuario el correo electrónico con los datos sobre la validación de su programa, finalizando así el ciclo de la aplicación.

6.5 Diagrama de despliegue

Como se ha visto anteriormente, la plataforma posee distintos bloques distribuidos en diferentes ámbitos. Por un lado, en el lado del usuario, ha de haber un sistema que permita acceder a la plataforma, como por ejemplo un navegador web. Después, en el lado del servidor, se encuentra toda la lógica de presentación y de negocio desarrollada en el proyecto que se va a realizar. Finalmente, fuera del servidor de la aplicación, se encuentra el servidor de correo, que es totalmente independiente del proyecto. En la figura 18 se puede ver un diagrama explicativo donde se ven estos tres elementos que forman parte de la arquitectura.

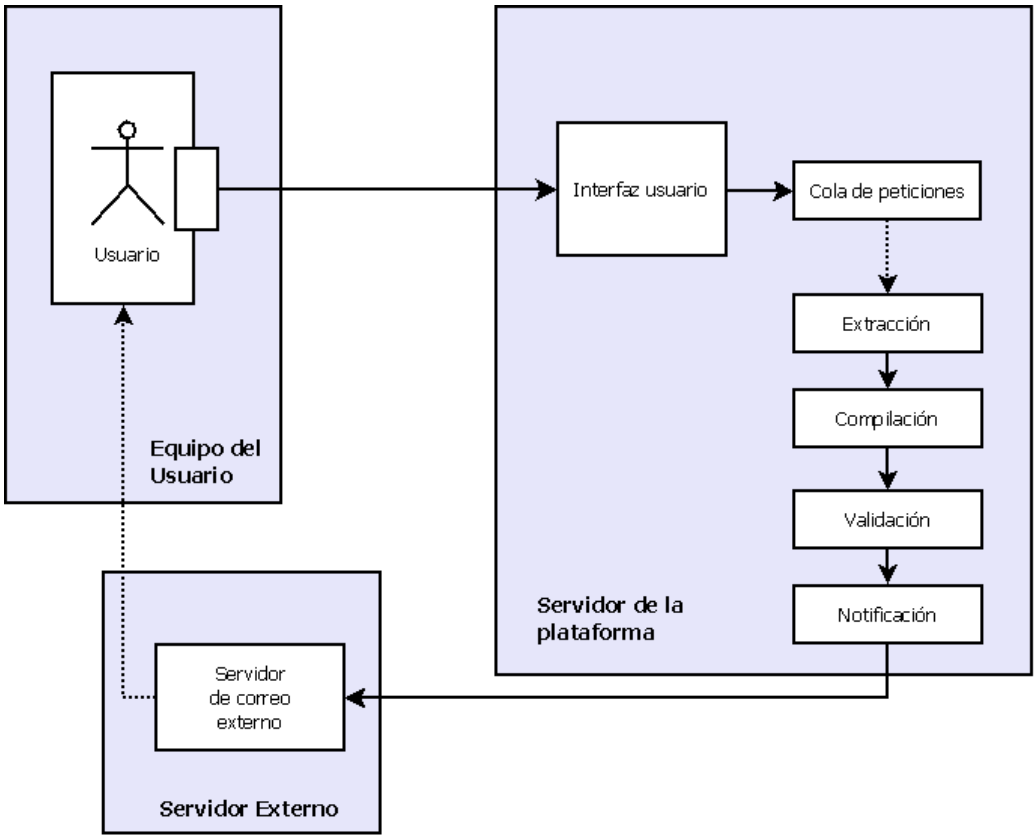

Figura 18: Diagrama de despliegue de la aplicación

6.6 Resumen y conclusiones

Se ha diseñado la aplicación con un único caso de uso que cumple con el objetivo principal de dar al usuario la opción de validar su código C vía web. En dicho caso de uso intervienen todos y cada uno de los módulos diseñados, que son: acceso web, gestión de peticiones, extractor, compilador, ejecutor y gestor de correos.

El usuario aportará la información necesaria, que son los datos para devolver el resultado, el fichero comprimido con el código fuente y las opciones de compilación y ejecución. Con esa información, la plataforma será capaz de generar el resultado de la validación y hacérselo llegar mediante un servidor de correo externo.

Se concluye pues, que desarrollando una plataforma con este diseño se cumplirá con el subobjetivo de ensamblar el software Valgrind con JEE.

Capítulo 7

Aspectos de implementación

Una vez definido el diseño de la plataforma, se procede a la implementación de la misma. Se quiere implementar de tal modo que, cumpliendo con el diseño del capítulo 6, se satisfaga el subobjetivos de ensamblar la tecnología JEE con el software Valgrind para ofrecer al usuario sus servicios remotamente.

En este capítulo se describen más técnicamente los módulos de la aplicación así como detalles de más bajo nivel relacionados directamente con la programación.

7.1 Esquema de implementación

El esquema, como se puede ver en la figura 19, se divide en cuatro grandes bloques:

- **Presentación y tratamiento de la información**

  Se empaqueta en un fichero WAR como establece la especificación de Java EE. Se encarga de ofrecer al usuario la interfaz para que introduzca sus datos y estos sean recogidos y analizados para comprobar que sean coherentes. Utiliza JSPs y un servlet que serán descritos más adelante.

- **Gestor de peticiones**

  Es un *Message-Driven Bean* que utiliza JMS para implementar la cola de peticiones.

- **Lógica de negocio**

  Este bloque se encarga de procesar el código del usuario para llegar al resultado final. Esta compuesto por un *Stateless Session Bean* que se encarga de ir a llamando a los componentes para realizar las distintas tareas. Estos componentes, implementados como clases Java, son el extractor, compilador, ejecutor y gestor de correos.

- **Usuario**

  En el bloque del usuario se encuentran el navegador web desde el que se accederá a la aplicación y el cliente de correo que recibirá el resultado de la validación. No dependen de la plataforma aquí desarrollada.

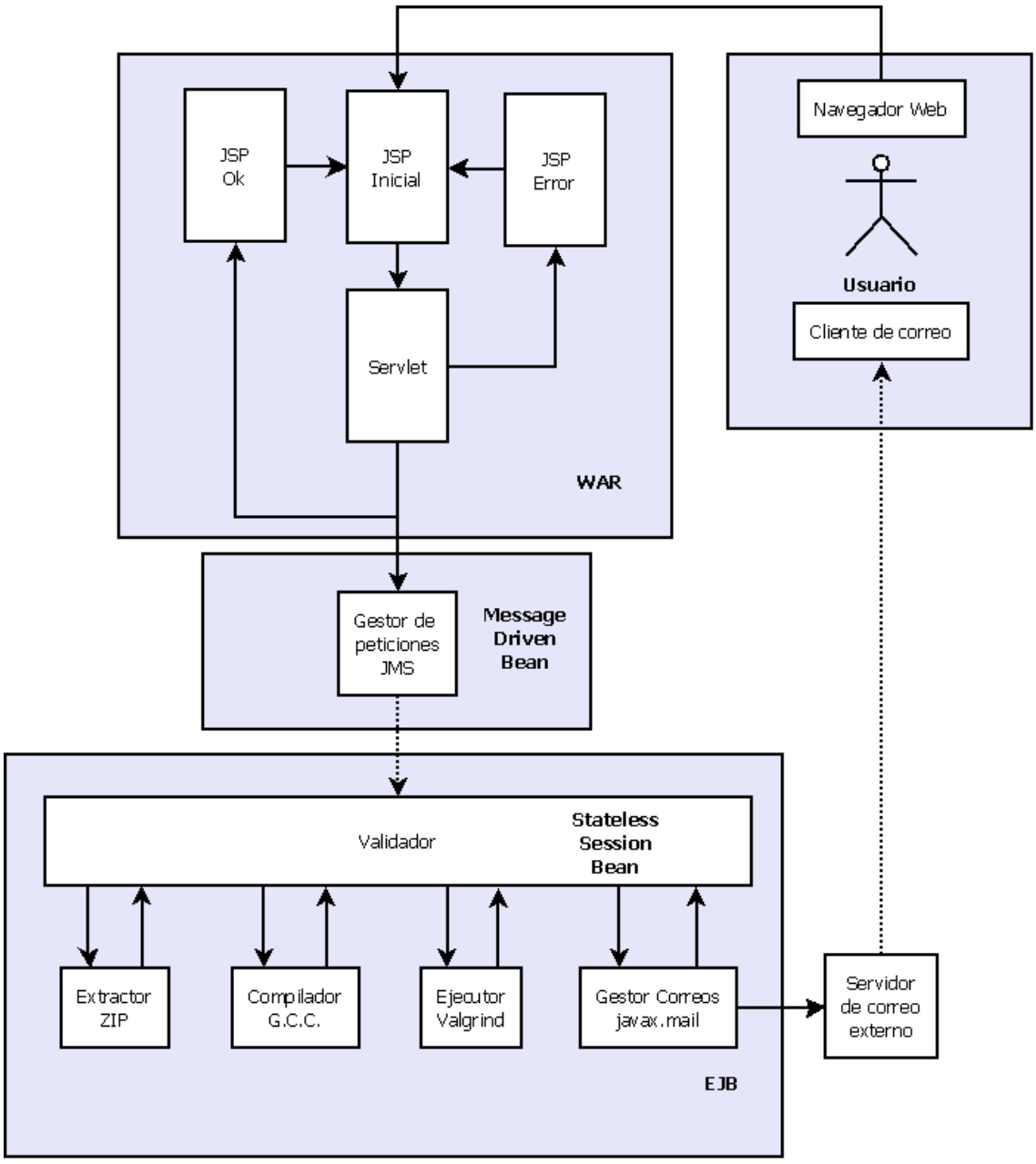

Figura 19: Esquema de implementación de la plataforma

En este capítulo se analizarán en detalle los distintos componentes de los tres bloques pertenecientes a la plataforma y la forma en que se almacenan los ficheros de las peticiones en el servidor.

7.1.1 Almacenamiento de ficheros de peticiones

Cuando el servidor recibe una petición, tiene que guardar la información que el usuario le ha facilitado y organizar una estructura de directorios donde almacenar los distintos recursos que se generan en el procesado del código del usuario. El esquema de esta estructura se ve en la figura 20.

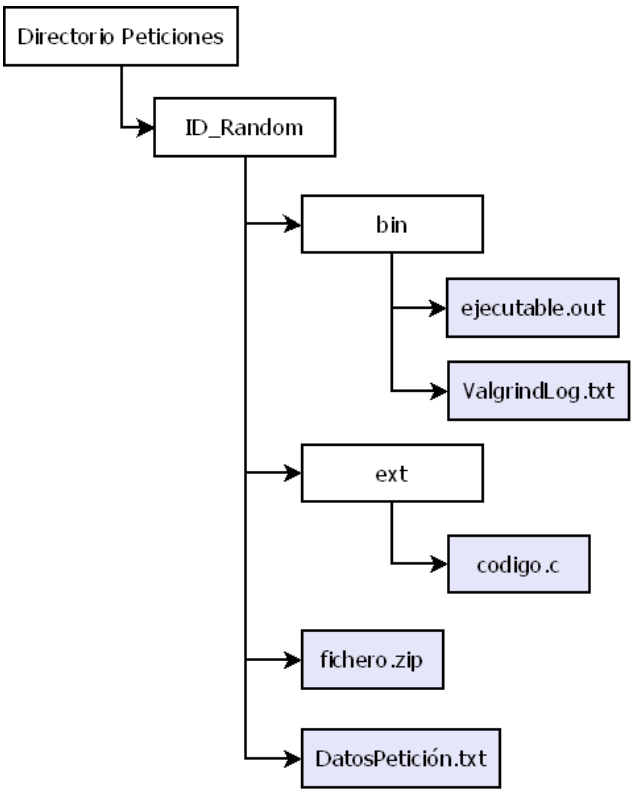

Figura 20: Árbol de directorios de peticiones

Para empezar, habrá un directorio en el servidor donde se guardarán todas las peticiones que se reciban. De él colgará un directorio por cada petición recibida y tendrán como nombre el ID del usuario más un número aleatorio entre 1 y 1000, reduciendo así las probabilidades de que dos directorios coincidan en el nombre.

Dentro del directorio de cada petición se guardará el fichero Zip enviado por el usuario y un fichero de texto con los datos de la petición (fecha y hora, nombre y correo del usuario, nombre del fichero zip, modo de uso, opciones de compilación y opciones de ejecución). Además habrá un directorio *ext* donde se guardará el código C extraído del fichero comprimido y un directorio *bin* donde se guardarán el fichero ejecutable y un fichero de texto con el resultado de Valgrind.

7.2 Presentación y tratamiento de la información

7.2.1 Acceso a la aplicación

El acceso a la aplicación se hace mediante un cliente web. Éste incluye un grupo de tres páginas JSP; la principal tiene un formulario para introducir la información; las otras sirven para informar al usuario de que la petición se ha hecho con éxito o para advertirle de que ha habido algún error. Los errores pueden deberse a que faltan datos en la petición o a que ha habido un error interno en el servidor.

A continuación vemos ejemplos de estas páginas.

Figura 21: Vista de la página inicial del acceso web

Como se puede observar en la figura 21, hay tres modos de uso:

- **Memcheck**: este modo utiliza la herramienta Memcheck de la suite Valgrind para analizar el programa en busca de problemas de memoria. No es necesario especificar ningún detalle más, salvo las opciones de compilación si hicieran falta.
- **Helgrind**: este modo hace uso de la herramienta Helgrind para detectar problemas de concurrencia en el programa analizado. Si se elige este modo es que se está usando programación multihilos, luego en las opciones de compilación habrá que poner como mínimo `-lpthread`.
- **Experto**: este modo brinda al usuario la oportunidad de utilizar la suite Valgrind como si estuviera instalada en su propio equipo. Al elegir este modo se activará una caja de texto usando código *javascript* ([27]), como se puede ver en la figura 22, en la que el usuario deberá escribir el comando con el que quiere ejecutar Valgrind, pudiendo elegir la herramienta, opciones y argumentos que desee.

Figura 22: Activación de la caja de texto en el modo Experto

La tecnología *javascript* también es utilizada en esta aplicación para comprobar que el fichero que se intenta enviar al servidor tiene formato zip. Si se intenta introducir otro tipo de archivo se avisará al usuario con la ventana de información que se ve en la figura 23. En caso de que el explorador utilizado por el usuario no soporte javascript y el acceso web permita enviar un fichero que no sea zip, será el módulo de extracción el que detecte el error.

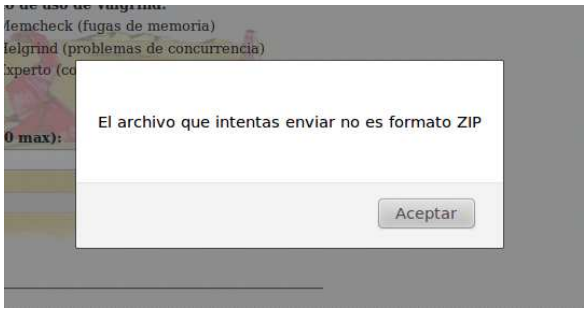

Figura 23: Mensaje de error generado por javascript

Si el usuario olvidara introducir alguno de los datos necesarios como son el identificador, la dirección de respuesta y/o la línea de ejecución si ha escogido el modo experto, se le advertirá mostrándole la página de la figura 24.

Figura 24: Advertencia de error introduciendo los datos

Cuando todo el proceso ha funcionado correctamente, se presentará al usuario una página como la de la figura 25. En ella se detallan los datos de la petición.

Figura 25: Confirmación de petición enviada al servidor

7.2.2 Servlet de recepción

Este componente se encarga de recibir los datos del usuario codificados en la petición *post* (HTTP) enviada por el formulario inicial. Una vez comprobada la información, si es coherente, se envía al módulo de encolado para esperar allí a ser procesada.

En la figura 26 se muestra el funcionamiento más detallado del código del módulo de acceso. Como se puede ver, el JSP inicial hace uso de Javascript para ver si el fichero adjuntado por el usuario es Zip, si es así pasa al servlet y si no lo indica al usuario para que rectifique.

El servlet se encarga de comprobar de que no falten datos como el ID, el correo electrónico, o el comando de ejecución en caso de haber elegido el modo Experto. Si todo es coherente enviará al usuario a un JSP informando del envío de la petición y enviará dicha petición a la cola. Si la información no es completa, se envía al usuario a un JSP indicándole el problema y dándole la oportunidad de que vuelva a intentarlo.

El servlet, por otro lado, se encarga de crear el directorio para almacenar la petición, guardar en él el fichero comprimido y crear un fichero de texto con los datos.

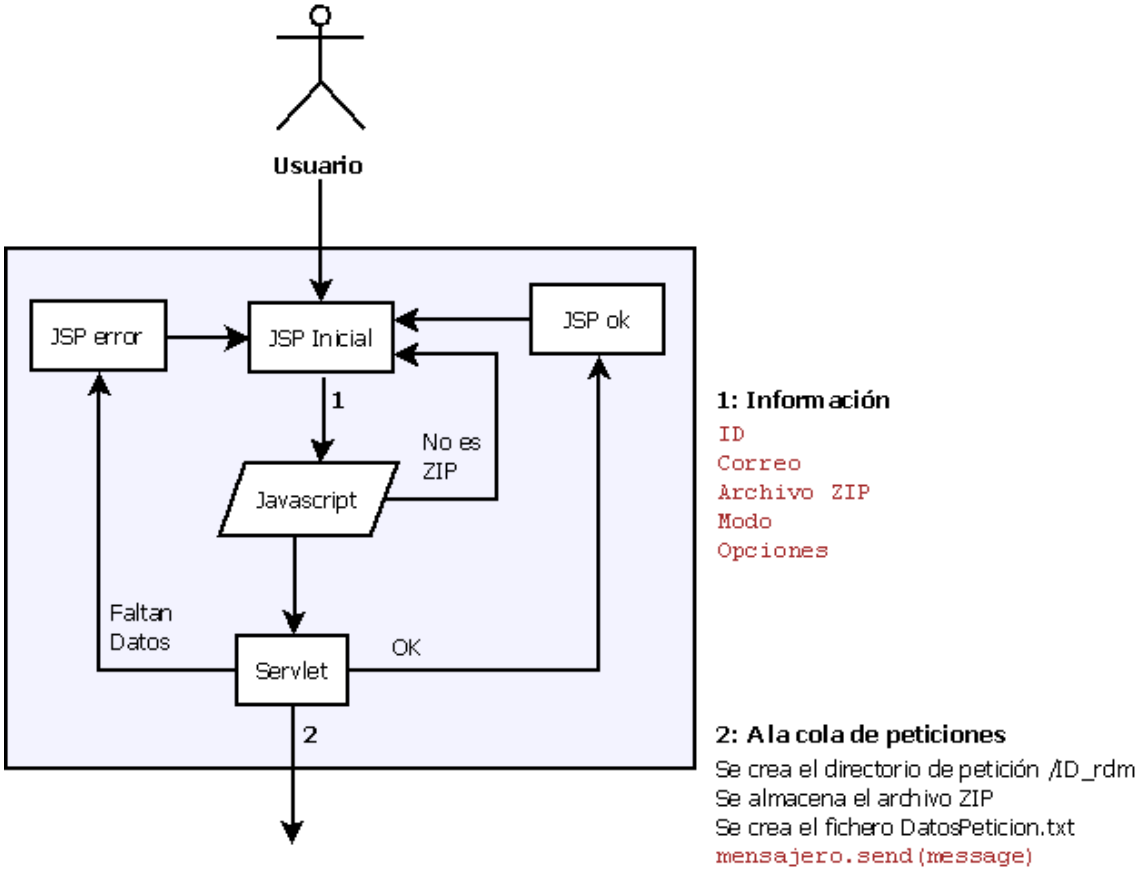

Figura 26: Diagrama de flujo del módulo de acceso

7.3 Gestor de peticiones

Está implementado utilizando la tecnología JMS que aporta JEE y facilita la gestión de colas. Cuando el sistema está saturado y no puede aceptar más peticiones temporalmente, la cola evita negar el servicio al usuario. Lo que hace es almacenar la petición hasta que pueda ser procesada.

Cuando llega el momento, la petición es enviada al siguiente módulo, denominado Validador. En este momento, la función de la cola para esa petición concreta llega a su fin.

En la figura 27 se observa el diagrama de flujo de este subcomponente. Se observa que cuando llega el momento procesa los datos del mensaje y llama al método correspondiente del módulo encargado de procesar la información.

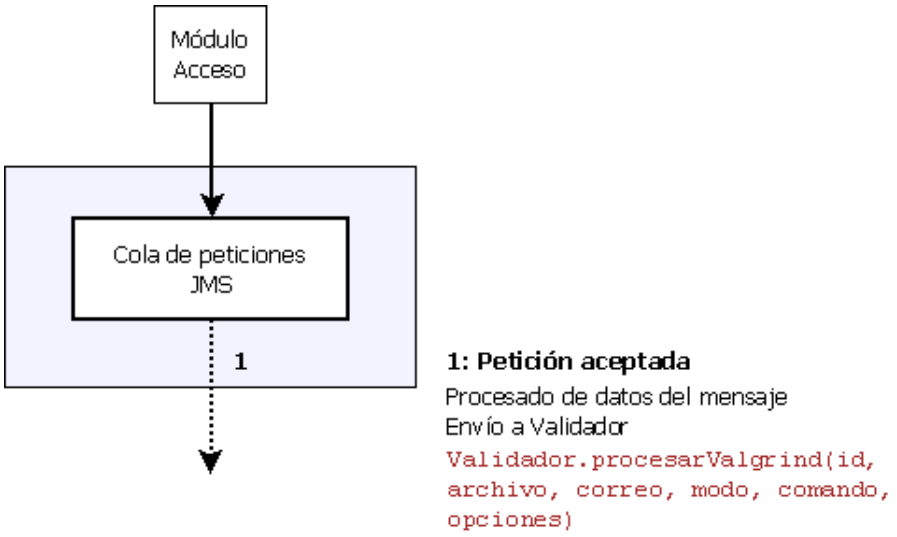

Figura 27: Diagrama de flujo de la cola de peticiones JMS

7.4 Lógica de negocio

7.4.1 Validador

Como hemos visto en la figura 19, este modulo está implementado como un *Stateless Session Bean* de Java llamado `Validador.java` cuya interfaz se muestra en la figura 28, y es el encargado de gestionar el procesado de la petición. Ese procesado se muestra en la figura 29.

Validador

```
boolean procesarValgrind (String
id, String archivo, String
correo, String modo, String
comando, String compilacion)
```

Figura 28: Clase `Validador`

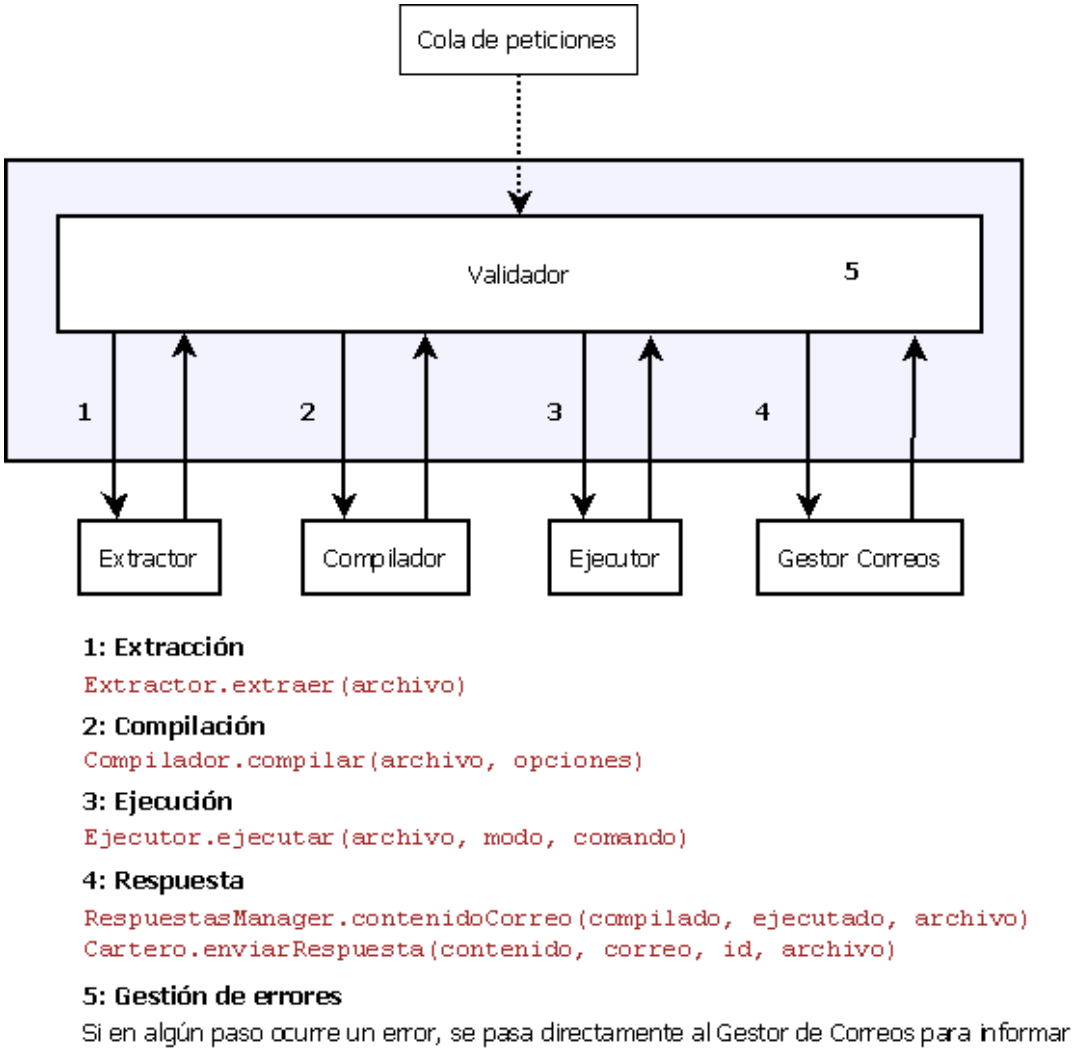

Figura 29: Diagrama de flujo del módulo Validador

- En primer lugar hace una llamada al extractor para que el archivo comprimido en zip que el usuario envió pase a ser un directorio con el código fuente del programa. Este código tiene que ser accesible por los siguientes módulos y se almacenará en un directorio específico.
- Después llama al compilador para que, utilizando el código fuente extraído anteriormente, genere un archivo ejecutable.
- Seguidamente llama al ejecutor, que es el encargado de ejecutar la herramienta de Valgrind especificada sobre el archivo ejecutable generado.
- Por último, hará una llamada al gestor de correos. Tanto si ha habido un error durante el procesado, en cuyo caso habrá pasado aquí directamente, como si el proceso ha sido satisfactorio, el gestor de correos será el encargado de enviar el informe a la dirección que indicó el usuario.

7.4.2 Extractor

Éste módulo recibe del Validador los archivos comprimidos y los descomprime en el directorio indicado. En la versión inicial de la plataforma solo se pueden descomprimir archivos *.zip*, pero debido a la estructura de la implementación es trivial añadir módulos para la extracción de otro tipo de archivos.

La implementación de este módulo consta de una clase Java llamada `ExtractorManager.java` cuya interfaz se muestra en la figura 30, que se encarga de comprobar el tipo de archivo, y otra llamada `ExtractorZip.java` cuya interfaz se muestra en la figura 31, que descomprime los ficheros en el directorio especificado.

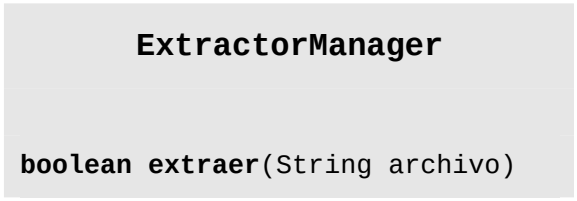

Figura 30: Clase `ExtractorManager`

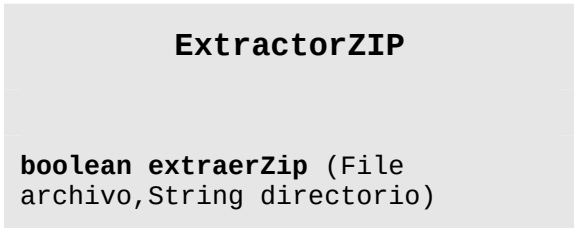

Figura 31: Clase `ExtractorZIP`

En la figura 32 se puede ver un diagrama de flujo para entender mejor las acciones que lleva a cabo este módulo.

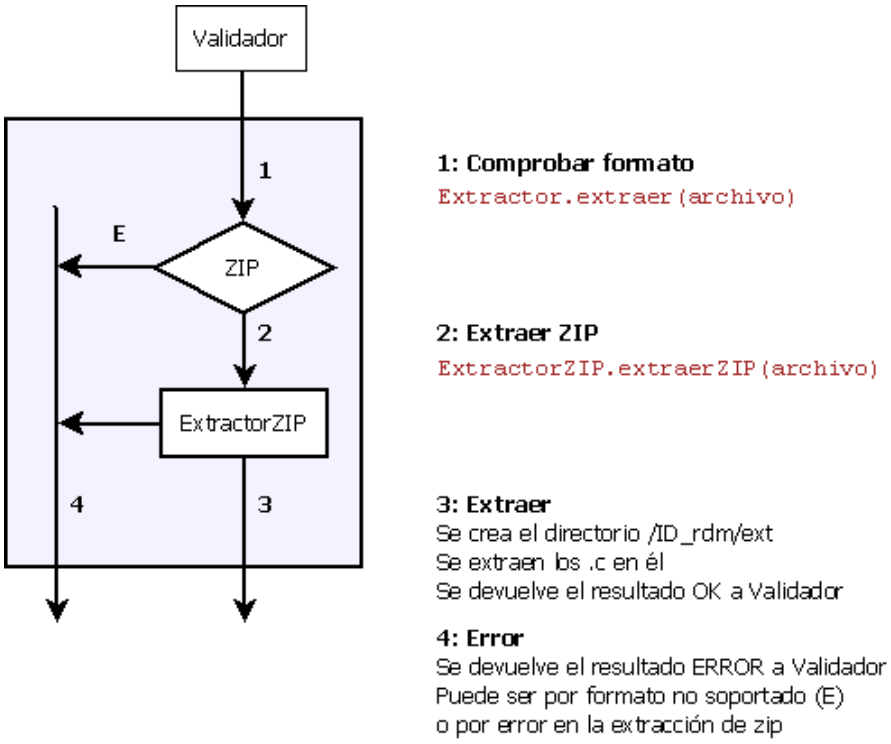

Figura 32: Diagrama de flujo del módulo Extractor

Como se puede ver, lo primero que hace es comprobar el tipo de fichero y después llamar al extractor correspondiente. En este caso solo hay extractor ZIP, pero por la estructura modular se pueden añadir otros extractores con pequeños cambios en el código. Si no hay extractor para el tipo de fichero, se devolverá error.

En este módulo se crea el fichero *ext* y los ficheros .c encontrados en el ZIP.

7.4.3 Compilador

Cualquier herramienta de la suite Valgrind necesita tener el fichero ejecutable del programa a analizar. Por lo tanto hay que compilar el código fuente obtenido para llegar a ese punto. Además hay que hacerlo con las opciones adecuadas, como pueden ser la de depuración (obligatoria) o la de programación multihilos (opción del usuario).

Este módulo, implementado como una clase Java llamada `Compilador.java` cuya interfaz se muestra en la figura 33, ejecuta un comando en un terminal de Linux en el que se utiliza la herramienta `gcc` con la opción de depuración `-d`, se incluyen todos los ficheros `.c` encontrados en el directorio de la petición y las opciones adicionales que el usuario a indicado.

El resultado será un fichero `Ejecutable.out` almacenado en un directorio específico. En caso de error de compilación, se creará un fichero de texto con la salida del comando `gcc` que será enviado al usuario como informe.

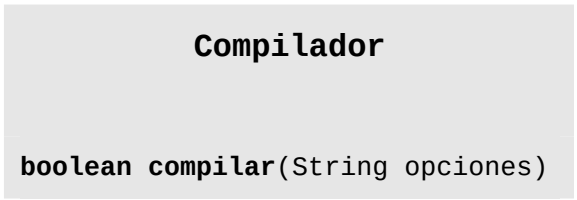

Figura 33: Clase `Compilador`

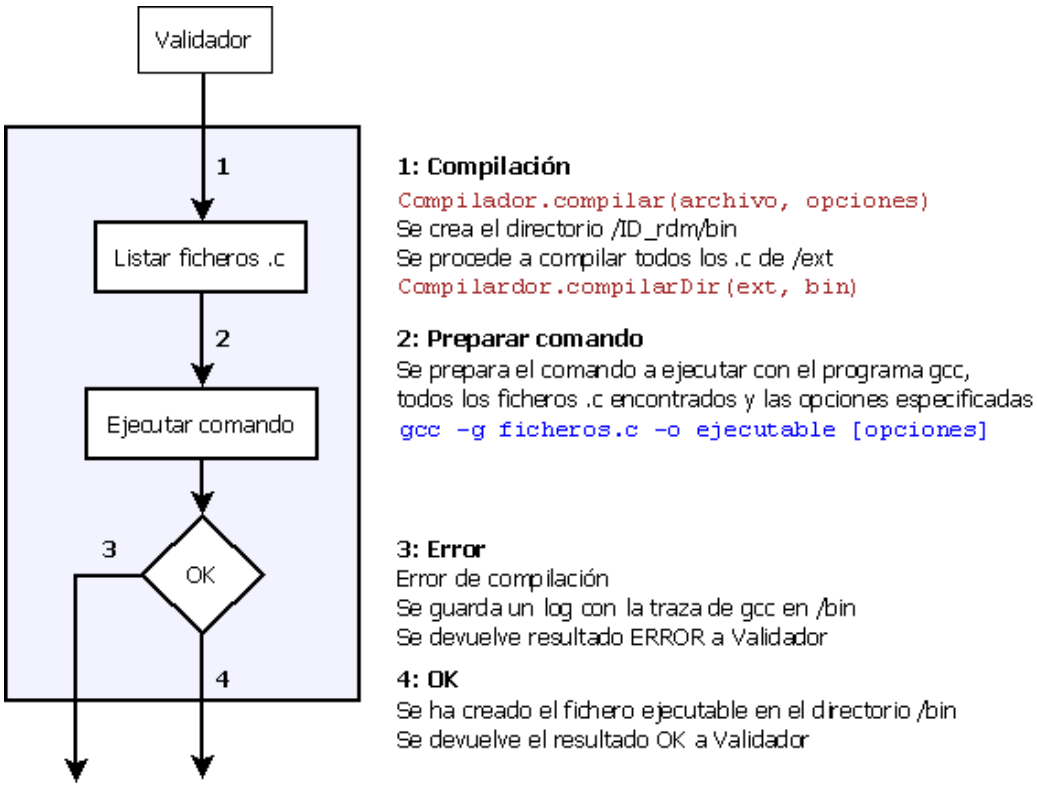

Figura 34: Diagrama de flujo del módulo Compilador

En la figura 34 se ve como el validador llama al módulo compilador pasándole como parámetros la ubicación de los archivos a compilar y las opciones de compilación. Después, se crea el directorio *bin* donde se guardará el ejecutable y se procede a compilar con GCC. Si hay error, se guarda un fichero de texto con la traza devuelta por gcc en el directorio *bin*, si todo sucede correctamente se crea el ejecutable y se devuelve resultado OK al validador.

7.4.4 Ejecutor

Este módulo está implementado por la clase `Ejecutor.java` cuya interfaz se muestra en la figura 35 y ejecuta en un terminal de Linux el comando correspondiente con el fichero `Ejecutable.out`. Dicho comando es compuesto en el código del módulo

atendiendo al modo de ejecución elegido y a la línea introducida por el usuario en el formulario inicial. El diagrama de flujo se encuentra en la figura 36.

Ejecutor

```
int ejecutar(String
nombreArchComp, String modo,
String comando)
```

Figura 35: Clase `Ejecutor`

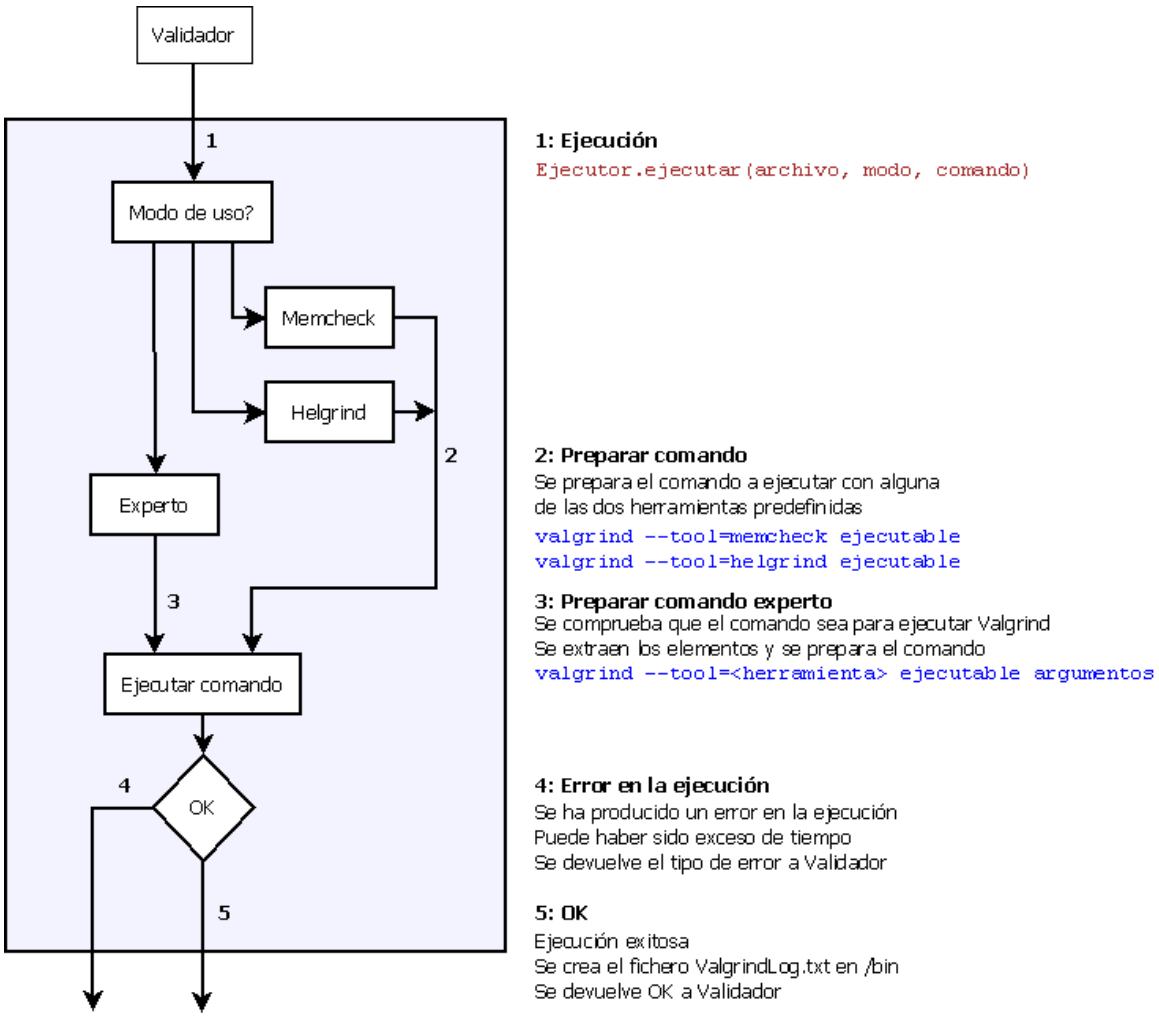

Figura 36: Diagrama de flujo del módulo Ejecutor

Si el modo es uno de los preestablecidos, Memcheck o Helgrind, se ejecutará el comando siguiente: `valgrind –tool=name Ejecutable`. (Donde `name` estaría el nombre de la herramienta).

Si el modo es Experto, se analizará la línea de ejecución introducida por el usuario, comprobando por cuestión de seguridad que lo que se quiere ejecutar es

Valgrind y no otro comando. Con este modo el usuario puede explotar todas las posibilidades que ofrece la suite Valgrind.

Además, este módulo dispone de un temporizador para evitar saturar el servidor con programas que tienen problemas como bucles infinitos. El límite de tiempo es un parámetro configurable por el administrador según sus requisitos. Si dicho tiempo se agota, la ejecución de Valgrind será abortada y se generará el consiguiente informe de error.

Como resultado de la ejecución se obtiene un fichero de texto que será enviado al usuario por correo electrónico. Dicho fichero contendrá el tiempo de ejecución empleado y la salida del comando ejecutado, ya sea el resultado de Valgrind sobre el programa del usuario, o un error de ejecución como, por ejemplo, un comando mal escrito en el modo Experto.

7.4.5 Gestor de correos

Este módulo es el último de la plataforma. Está implementado con la clase `Cartero.java` cuya interfaz se muestra en la figura 37 y es el encargado de enviar al servidor de correo externo el email con la dirección del usuario. Utiliza la librería `javax.mail` de Java.

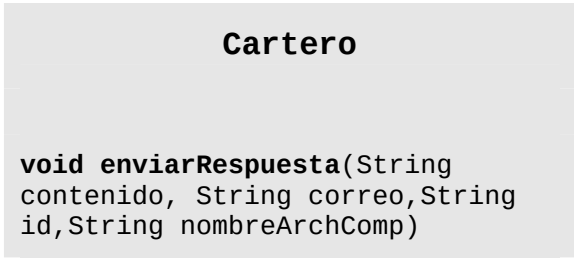

Figura 37: Clase `Cartero`

El contenido se crea mediante una clase llamada `RespuestasManager.java` cuya interfaz se muestra en la figura 38, que establece lo que se le va a mandar al usuario exactamente. Los casos con el contenido correspondiente son los siguientes:

- Error de extracción
  Aviso de error en la extracción
- Error de compilación
  Log con la traza de error de GCC
- Error de ejecución
  Aviso de error en la ejecución
- Tiempo excedido
  Aviso de tiempo de ejecución excedido

- Informe de Valgrind
  Log con el resultado de Valgrind

RespuestasManager

```
String contenidoCorreo(boolean
compilado, int ejecutado, String
archivo)
```

Figura 38: Clase `RespuestasManager`

En la figura 39 se observa un diagrama de flujo del módulo. En él se aprecia cómo se comprueba el estado del procesado, que puede ser completo (ejecutado correctamente) o que haya habido errores en las distintas partes, para después generar la respuesta correspondiente y enviarla al usuario mediante el servidor de correo externo.

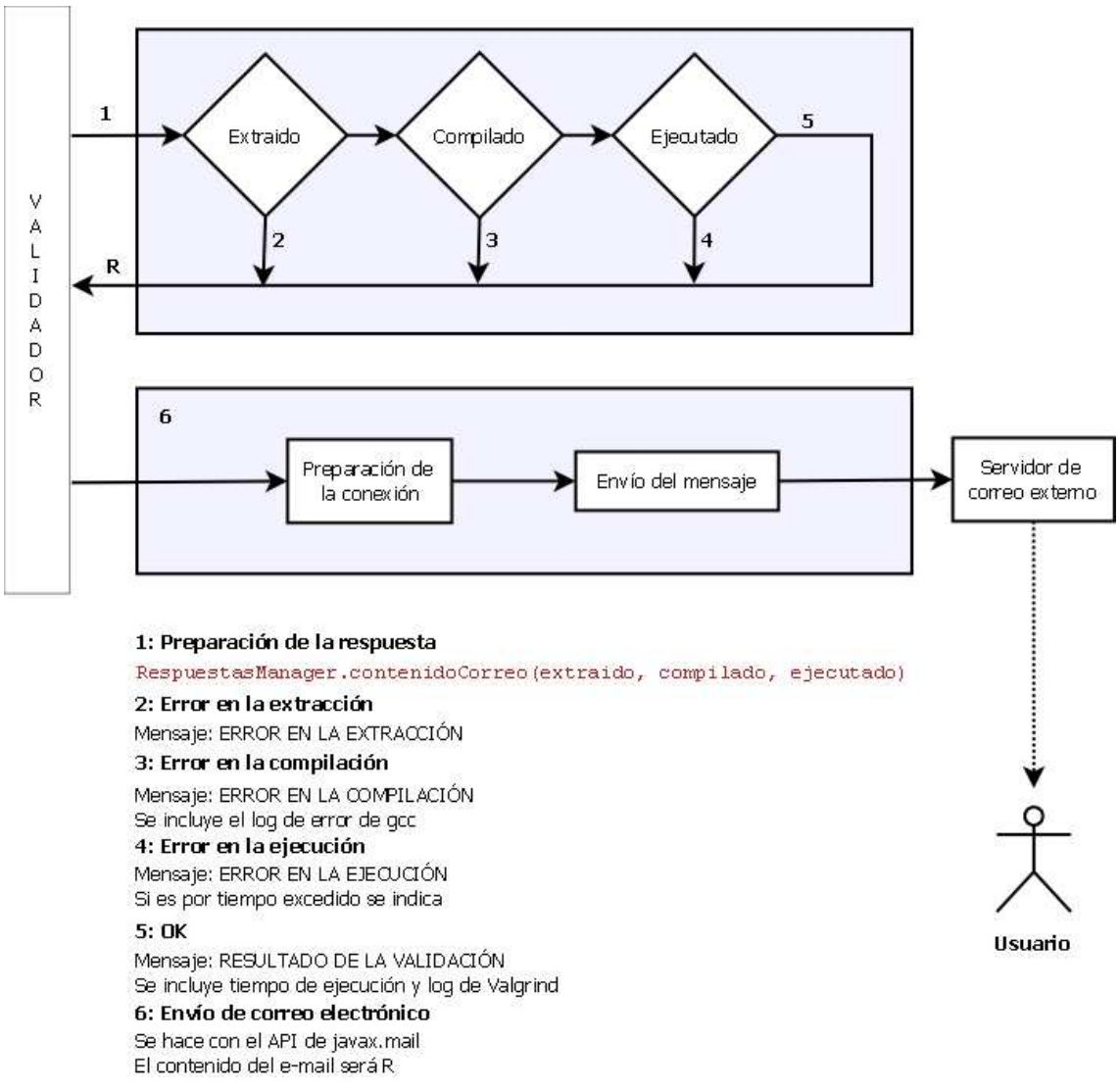

Figura 39: Diagrama de flujo del módulo de Gestión de Correos

7.10 Resumen y conclusiones

Se han implementado todos los componentes como se habían diseñado en el capítulo 6. Se parte de tres grandes bloques y se acaba implementado cada subcomponente encargado de una tarea concreta.

Como resultado de la implementación de los componentes descritos en este capítulo, se tiene una plataforma que cumple con los objetivos del proyecto. Dicha plataforma es capaz de ofrecer al usuario un acceso web donde hacer la petición con diversas opciones, gestionar las múltiples peticiones que pueda tener, extraer los archivos, compilarlos y ejecutarlos con las opciones del usuario, gestionar los distintos tipos de errores posibles y, finalmente, enviar un informe al usuario.

Ahora se deben diseñar y llevar a cabo unas pruebas para comprobar el rendimiento y el alcance que puede tener la plataforma desarrollada. Se hará en el capítulo 8..

Capítulo 8

Pruebas de rendimiento

En este capítulo se analizará si la aplicación desarrollada es viable o no. Para ello se dispondrán unos escenarios de prueba, se verá el resultado y se llegará a una conclusión.

8.1 Estado actual de la implementación

La aplicación está implementada y montada sobre un servidor Glassfish. Las peticiones las recibe un servlet que después las envía a la cola JMS. Un MDB va atendiendo las peticiones y enviándolas a un *Stateless Session Bean* que se encarga de ir llamando a los diferentes módulos para descomprimir, compilar, ejecutar y enviar el resultado.

8.2 Escenarios de prueba

Para llevar a cabo las pruebas de rendimiento se van a utilizar 10 programas en código C que utilizan el estándar POSIX de programación concurrente ([28]). Estos programas se han obtenido del *Lawrence Livermore National Laboratory*, un centro de computación de alto rendimiento de EEUU. Son programas que se pueden encontrar en el apartado de tutoriales y ejemplos de su página web y están orientados al aprendizaje.

Estos programas son interesantes para realizar estas pruebas porque poseen diversos tipos de errores. Dichos errores están documentados y se tiene también la solución. Lo que van a aportar a las pruebas de la plataforma es que se va a ver que tipos de problemas es capaz de detectar, y cual es el resultado que ofrece la herramienta utilizada de Valgrind en cada situación.

En este apartado se incluye la tabla 3 donde se indican las características de todos los programas, su tiempo de ejecución con o sin herramientas y algunas observaciones. Además, también se puede ver para cada programa el resultado que ofrecen tanto Memcheck como Valgrind y un análisis del resultado.

Tabla 3: Programas de ejemplo para las pruebas

Programa	Descripción del problema/solución	Observaciones	Tiempo de ejecución (en segundos)		
			Simple	Memcheck	Helgrind
Bug1.c	Hay más de un hilo esperando una señal y solo la recibe uno de ellos. El programa se queda bloqueado.		-	-	-
Bug1fix.c	Soluciona el problema de Bug1.c con una señal broadcast para que todos los hilos continúen.		10,005	10,560	10,771
Bug2.c	Se realizan lecturas y escrituras fuera del espacio de memoria reservado al hilo.		3,051	5,820	7,745
Bug2fix.	Se solucionan los problemas de Bug2.c reservando adecuadamente el espacio de memoria.		3,045	5,870	7,729
Bug3.c	Se utiliza una manera insegura e incorrecta de pasar el argumento a la rutina de creación del hilo.		1,003	3,140	1,779
Bug4.c	No hay sincronización entre la señal de espera y la señal de continuar. El programa probablemente se quedará bloqueado.	Compilar con la opción `-lm`	-	-	-
Bug4fix.c	Se soluciona el problema de Bug4.c con una sincronización correcta, aunque no eficiente del todo.	Compilar con la opción `-lm`	1,192	3,070	3,805
Bug5.c	Se finaliza el programa sin esperar a que el hilo haya acabado su ejecución.	Compilar con la opción `-lm`	0,003	2,090	5,127
Bug6.c	Varios hilos leen y escriben en la misma zona de memoria sin ninguna sincronización. El valor final no se puede predecir.		0,027	1,050	6,781
Bug6fix.c	Soluciona la condición de carrera de Bug6.c mediante sincronización con mutex.		0,017	1,000	7,022

Hay que tener en cuenta que el tiempo que incrementan las herramientas depende del tamaño del programa y del número de errores encontrados. Por eso en algunos de estos ejemplos puede no verse claramente la diferencia (Bug1fix.c). Sin embargo, en el resto se detectan variaciones de tiempo considerables, como cabía esperar.

También hay que indicar que para los programas Bug4, Bug4fix y Bug5 hay que utilizar la opción de compilación `-lm`, ya que hacen uso de una librería externa y hay que incluirla para que funcione correctamente.

8.2.1 Bug1.c

Este ejemplo contiene cuatro hilos que durante su ejecución esperan una señal para continuar. El problema radica en que la señal solo la recibe uno de ellos, luego el programa nunca acabará porque tendrá tres hilos sin poder continuar y finalizar su ejecución.

Si se envía este fichero a la plataforma desarrollada, no se obtendrá ningún resultado, puesto que superará siempre el tiempo máximo de ejecución. Sin embargo, ejecutándolo desde el propio equipo podemos suspender el proceso cuando se haya quedado bloqueado y obtendremos los siguientes resultados:

Para **Memcheck**:

```
==3548== HEAP SUMMARY:
==3548==     in use at exit: 544 bytes in 4 blocks
==3548==   total heap usage: 11 allocs, 7 frees, 1,798 bytes allocated
==3548==
==3548== LEAK SUMMARY:
==3548==    definitely lost: 0 bytes in 0 blocks
==3548==    indirectly lost: 0 bytes in 0 blocks
==3548==      possibly lost: 544 bytes in 4 blocks
==3548==    still reachable: 0 bytes in 0 blocks
==3548==         suppressed: 0 bytes in 0 blocks
==3548== Rerun with --leak-check=full to see details of leaked memory
==3548==
==3548== For counts of detected and suppressed errors, rerun with: -v
==3548== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 15 from 8)
```

Para **Helgrind**:

```
==3569== For counts of detected and suppressed errors, rerun with: -v
==3569== Use --history-level=approx or =none to gain increased speed, at
==3569== the cost of reduced accuracy of conflicting-access information
==3569== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 588 from 105)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se

puede observar que efectivamente la utilización de la memoria se hace de forma correcta. Entonces se concluye que Memchek es útil para el proyecto en este caso.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. El programa analizado tiene un error utilizando los hilos, ya que tres de ellos se quedan esperando infinitamente, y la herramienta no lo ha detectado. Por esta razón, en este caso Helgrind no aporta información que sería importante.

8.2.2 Bug1fix.c

Este ejemplo corrige al del apartado anterior. La solución es enviar una señal en modo broadcast, de forma que todos los hilos que están esperando puedan continuar y finalizar su ejecución.

Los resultados obtenidos de enviar esta aplicación a la plataforma son los siguientes:

Para **Memcheck**:

```
==2934== HEAP SUMMARY:
==2934==     in use at exit: 982 bytes in 5 blocks
==2934==   total heap usage: 11 allocs, 6 frees, 1,798 bytes allocated
==2934==
==2934== LEAK SUMMARY:
==2934==    definitely lost: 0 bytes in 0 blocks
==2934==    indirectly lost: 0 bytes in 0 blocks
==2934==      possibly lost: 0 bytes in 0 blocks
==2934==    still reachable: 982 bytes in 5 blocks
==2934==         suppressed: 0 bytes in 0 blocks
==2934== Rerun with --leak-check=full to see details of leaked memory
==2934==
==2934== For counts of detected and suppressed errors, rerun with: -v
==2934== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 15 from 8)
```

Para **Helgrind**:

```
==3317== For counts of detected and suppressed errors, rerun with: -v
==3317== Use --history-level=approx or =none to gain increased speed, at
==3317== the cost of reduced accuracy of conflicting-access information
==3317== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 593 from 95)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se puede observar que efectivamente la utilización de la memoria se hace de forma correcta. Entonces se concluye que Memchek es útil para el proyecto en este caso.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. En este caso el resultado es el esperado, puesto que el programa se ejecuta sin ningún tipo de problema de bloqueo.

8.2.3 Bug2.c

Este ejemplo hace un mal uso de la memoria asignada al programa. No reserva explícitamente el espacio de memoria que luego usa.

Los resultados obtenidos de enviar esta aplicación a la plataforma son los siguientes:

Para **Memcheck**:

```
==2952== Warning: client switching stacks?  SP change: 0x49b7398 --> 0x45e6a70
==2952==          to suppress, use: --max-stackframe=4000040 or greater
==2952== Thread 2:
==2952== Invalid write of size 4
==2952==    at 0x8048580: Hello (bug2.c:21)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b7388 is on thread 2's stack
==2952==
==2952== Warning: client switching stacks?  SP change: 0x65ba398 --> 0x61e9a70
==2952==          to suppress, use: --max-stackframe=4000040 or greater
==2952== Warning: client switching stacks?  SP change: 0x5db9398 --> 0x59e8a70
==2952==          to suppress, use: --max-stackframe=4000040 or greater
==2952==          further instances of this message will not be shown.
==2952== Invalid write of size 4
==2952==    at 0x804858F: Hello (bug2.c:23)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b738c is on thread 2's stack
==2952==
==2952== Invalid read of size 4
==2952==    at 0x80485A9: Hello (bug2.c:23)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b738c is on thread 2's stack
==2952==
==2952== Invalid read of size 4
==2952==    at 0x8048598: Hello (bug2.c:25)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b738c is on thread 2's stack
==2952==
==2952== Invalid read of size 4
==2952==    at 0x804859B: Hello (bug2.c:25)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b738c is on thread 2's stack
==2952==
==2952== Invalid read of size 4
==2952==    at 0x80485A5: Hello (bug2.c:23)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b738c is on thread 2's stack
==2952==
==2952== Invalid read of size 4
```

```
==2952==    at 0x80485B0: Hello (bug2.c:23)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b738c is on thread 2's stack
==2952==
==2952== Invalid write of size 8
==2952==    at 0x804859E: Hello (bug2.c:25)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b7148 is on thread 2's stack
==2952==
==2952== Invalid read of size 8
==2952==    at 0x80485B2: Hello (bug2.c:27)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b7380 is on thread 2's stack
==2952==
==2952== Invalid read of size 4
==2952==    at 0x80485BE: Hello (bug2.c:27)
==2952==    by 0x4040E98: start_thread (pthread_create.c:304)
==2952==    by 0x412573D: clone (clone.S:130)
==2952==  Address 0x49b7388 is on thread 2's stack
==2952==
==2952==
==2952== HEAP SUMMARY:
==2952==     in use at exit: 1,088 bytes in 8 blocks
==2952==   total heap usage: 13 allocs, 5 frees, 2,070 bytes allocated
==2952==
==2952== LEAK SUMMARY:
==2952==    definitely lost: 0 bytes in 0 blocks
==2952==    indirectly lost: 0 bytes in 0 blocks
==2952==      possibly lost: 1,088 bytes in 8 blocks
==2952==    still reachable: 0 bytes in 0 blocks
==2952==         suppressed: 0 bytes in 0 blocks
==2952== Rerun with --leak-check=full to see details of leaked memory
==2952==
==2952== For counts of detected and suppressed errors, rerun with: -v
==2952== ERROR SUMMARY: 2500077 errors from 10 contexts (suppressed: 15 from
8)
```

Para **Helgrind**:

```
 ==3335== Warning: client switching stacks?  SP change: 0x49b8360 -->
0x45e7a38
==3335==          to suppress, use: --max-stackframe=4000040 or greater
==3335== Warning: client switching stacks?  SP change: 0x55b9360 --> 0x51e8a38
==3335==          to suppress, use: --max-stackframe=4000040 or greater
==3335== Warning: client switching stacks?  SP change: 0x5dba360 --> 0x59e9a38
==3335==          to suppress, use: --max-stackframe=4000040 or greater
==3335==          further instances of this message will not be shown.
==3335==
==3335== For counts of detected and suppressed errors, rerun with: -v
==3335== Use --history-level=approx or =none to gain increased speed, at
==3335== the cost of reduced accuracy of conflicting-access information
==3335== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 346 from 21)
```

Análisis de resultados:

Memcheck indica que ha encontrado 2.500.077 errores en 10 contextos. Como se puede ver en la traza hay tres ocasiones en las que se producen escrituras no válidas y siete en las que se producen lecturas no válidas. En este caso se obtiene el resultado

esperado para el detector de problemas de memoria. Se concluye entonces que Memcheck es útil en este caso porque ayuda a encontrar ciertos errores.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. En este caso el resultado es el esperado, puesto que el programa se ejecuta sin ningún tipo de problema de bloqueo o condiciones de carrera.

8.2.4 Bug2fix.c

Este programa soluciona los problemas del anterior (Bug2.c). Lo hace reservando el espacio de memoria necesario para que al usarlo no de problemas.

Los resultados de enviar este programa a la plataforma son los siguientes:

Para **Memcheck**:

```
==3127== Warning: client switching stacks?  SP change: 0x4f3fed8 --> 0x4b6f5b0
==3127==          to suppress, use: --max-stackframe=4000040 or greater
==3127== Thread 3:
==3127== Invalid write of size 4
==3127==    at 0x80485C0: Hello (bug2fix.c:29)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fec8 is on thread 3's stack
==3127==
==3127== Warning: client switching stacks?  SP change: 0x467aed8 --> 0x42aa5b0
==3127==          to suppress, use: --max-stackframe=4000040 or greater
==3127== Warning: client switching stacks?  SP change: 0x5404ed8 --> 0x50345b0
==3127==          to suppress, use: --max-stackframe=4000040 or greater
==3127==          further instances of this message will not be shown.
==3127== Invalid write of size 4
==3127==    at 0x80485CF: Hello (bug2fix.c:31)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fecc is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x80485E9: Hello (bug2fix.c:31)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fecc is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x80485D8: Hello (bug2fix.c:33)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fecc is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x80485DB: Hello (bug2fix.c:33)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fecc is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x80485E5: Hello (bug2fix.c:31)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
```

```
==3127==  Address 0x4f3fecc is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x80485F0: Hello (bug2fix.c:31)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fecc is on thread 3's stack
==3127==
==3127== Invalid write of size 8
==3127==    at 0x80485DE: Hello (bug2fix.c:33)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fc88 is on thread 3's stack
==3127==
==3127== Invalid read of size 8
==3127==    at 0x80485F2: Hello (bug2fix.c:35)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3feb8 is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x80485FE: Hello (bug2fix.c:35)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fec8 is on thread 3's stack
==3127==
==3127== Invalid write of size 4
==3127==    at 0x4042860: pthread_attr_getstacksize
(pthread_attr_getstacksize.c:36)
==3127==    by 0x8048620: Hello (bug2fix.c:36)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fec4 is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x8048621: Hello (bug2fix.c:37)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fec4 is on thread 3's stack
==3127==
==3127== Invalid read of size 4
==3127==    at 0x804862D: Hello (bug2fix.c:37)
==3127==    by 0x4040E98: start_thread (pthread_create.c:304)
==3127==    by 0x412573D: clone (clone.S:130)
==3127==  Address 0x4f3fec8 is on thread 3's stack
==3127==
==3127==
==3127== HEAP SUMMARY:
==3127==     in use at exit: 1,088 bytes in 8 blocks
==3127==   total heap usage: 13 allocs, 5 frees, 2,070 bytes allocated
==3127==
==3127== LEAK SUMMARY:
==3127==    definitely lost: 0 bytes in 0 blocks
==3127==    indirectly lost: 0 bytes in 0 blocks
==3127==      possibly lost: 1,088 bytes in 8 blocks
==3127==    still reachable: 0 bytes in 0 blocks
==3127==         suppressed: 0 bytes in 0 blocks
==3127== Rerun with --leak-check=full to see details of leaked memory
==3127==
==3127== For counts of detected and suppressed errors, rerun with: -v
==3127== ERROR SUMMARY: 2500079 errors from 13 contexts (suppressed: 15 from
8)
```

Para **Helgrind**:

```
==3355== Warning: client switching stacks?  SP change: 0x467bea0 --> 0x42ab578
```

```
==3355==          to suppress, use: --max-stackframe=4000040 or greater
==3355== Warning: client switching stacks?  SP change: 0x4f40ea0 --> 0x4b70578
==3355==          to suppress, use: --max-stackframe=4000040 or greater
==3355== Warning: client switching stacks?  SP change: 0x5405ea0 --> 0x5035578
==3355==          to suppress, use: --max-stackframe=4000040 or greater
==3355==          further instances of this message will not be shown.
==3355==
==3355== For counts of detected and suppressed errors, rerun with: -v
==3355== Use --history-level=approx or =none to gain increased speed, at
==3355== the cost of reduced accuracy of conflicting-access information
==3355== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 349 from 19)
```

Análisis de resultados:

Memcheck indica que después del análisis ha encontrado 2.500.079 errores en 13 contextos; cuatro de escritura inválida y nueve de lectura inválida. No es el resultado esperado, puesto que en teoría este programa debía resolver los problemas de memoria del programa anterior. Para este caso concreto habría que revisar el código del programa porque se han encontrado errores no previstos.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. En este caso el resultado es el esperado, pues que el programa se ejecuta sin ningún tipo de problema de bloqueo ni condiciones de carrera.

8.2.5 Bug3.c

Este ejemplo muestra una forma insegura (incorrecta) de pasar los argumentos del hilo a la hora de crearlo. Los resultados devueltos por la plataforma son los siguientes:

Para **Memcheck**:

```
==2971== HEAP SUMMARY:
==2971==     in use at exit: 1,088 bytes in 8 blocks
==2971==   total heap usage: 13 allocs, 5 frees, 2,070 bytes allocated
==2971==
==2971== LEAK SUMMARY:
==2971==    definitely lost: 0 bytes in 0 blocks
==2971==    indirectly lost: 0 bytes in 0 blocks
==2971==      possibly lost: 1,088 bytes in 8 blocks
==2971==    still reachable: 0 bytes in 0 blocks
==2971==         suppressed: 0 bytes in 0 blocks
==2971== Rerun with --leak-check=full to see details of leaked memory
==2971==
==2971== For counts of detected and suppressed errors, rerun with: -v
==2971== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 15 from 8)
```

Para **Helgrind**:

```
==3376== For counts of detected and suppressed errors, rerun with: -v
==3376== Use --history-level=approx or =none to gain increased speed, at
==3376== the cost of reduced accuracy of conflicting-access information
==3376== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 344 from 19)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se puede observar que efectivamente la utilización de la memoria se hace de forma correcta.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. En este caso el resultado es el esperado, puesto que el programa se ejecuta sin ningún tipo de problema de bloqueo o condición de carrera.

8.2.6 Bug4.c

Este ejemplo muestra un bloqueo debido a la mala sincronización de los hilos. Como resultado, el programa nunca finaliza su ejecución, por lo que la plataforma no podría ofrecer ningún resultado. Sin embargo, ejecutándolo en el propio equipo y suspendiendo el proceso cuando se haya quedado bloqueado, Valgrind nos ofrece estos resultados:

Para **Memcheck**:

```
==3529== HEAP SUMMARY:
==3529==     in use at exit: 408 bytes in 3 blocks
==3529==   total heap usage: 8 allocs, 5 frees, 1,390 bytes allocated
==3529==
==3529== LEAK SUMMARY:
==3529==    definitely lost: 0 bytes in 0 blocks
==3529==    indirectly lost: 0 bytes in 0 blocks
==3529==      possibly lost: 408 bytes in 3 blocks
==3529==    still reachable: 0 bytes in 0 blocks
==3529==         suppressed: 0 bytes in 0 blocks
==3529== Rerun with --leak-check=full to see details of leaked memory
==3529==
==3529== For counts of detected and suppressed errors, rerun with: -v
==3529== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 17 from 8)
```

Para **Helgrind**:

```
==2545== For counts of detected and suppressed errors, rerun with: -v
==2545== Use --history-level=approx or =none to gain increased speed, at
==2545== the cost of reduced accuracy of conflicting-access information
==2545== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 238 from 58)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se

puede observar que efectivamente la utilización de la memoria se hace de forma correcta.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. No es el resultado esperado puesto que se sabe que el programa acaba con un bloqueo. Helgrind no es capaz de detectarlo así que no sería útil en este caso.

8.2.7 Bug4fix.c

Este ejemplo muestra una posible solución a la des-sincronización del ejercicio anterior. Ahora ya finaliza su ejecución, así que se puede obtener estos resultados enviándolo a la plataforma:

Para **Memcheck**:

```
==3017== HEAP SUMMARY:
==3017==     in use at exit: 982 bytes in 5 blocks
==3017==   total heap usage: 8 allocs, 3 frees, 1,390 bytes allocated
==3017==
==3017== LEAK SUMMARY:
==3017==    definitely lost: 0 bytes in 0 blocks
==3017==    indirectly lost: 0 bytes in 0 blocks
==3017==      possibly lost: 0 bytes in 0 blocks
==3017==    still reachable: 982 bytes in 5 blocks
==3017==         suppressed: 0 bytes in 0 blocks
==3017== Rerun with --leak-check=full to see details of leaked memory
==3017==
==3017== For counts of detected and suppressed errors, rerun with: -v
==3017== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 17 from 8)
```

Para **Helgrind**:

```
==2591== For counts of detected and suppressed errors, rerun with: -v
==2591== Use --history-level=approx or =none to gain increased speed, at
==2591== the cost of reduced accuracy of conflicting-access information
==2591== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 254 from 54)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se puede observar que efectivamente la utilización de la memoria se hace de forma correcta.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. Este sí sería un resultado esperado puesto que el programa se ejecuta correctamente sin ningún tipo de bloqueo.

8.2.8 Bug5.c

Este ejemplo muestra como el programa finaliza antes de que el hilo haya acabado su trabajo. Lo que las herramientas de Valgrind obtendrían sería esto:

Para **Memcheck**:

```
==3035== HEAP SUMMARY:
==3035==     in use at exit: 680 bytes in 5 blocks
==3035==   total heap usage: 5 allocs, 0 frees, 680 bytes allocated
==3035==
==3035== LEAK SUMMARY:
==3035==    definitely lost: 0 bytes in 0 blocks
==3035==    indirectly lost: 0 bytes in 0 blocks
==3035==      possibly lost: 680 bytes in 5 blocks
==3035==    still reachable: 0 bytes in 0 blocks
==3035==         suppressed: 0 bytes in 0 blocks
==3035== Rerun with --leak-check=full to see details of leaked memory
==3035==
==3035== For counts of detected and suppressed errors, rerun with: -v
==3035== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 15 from 6)
```

Para **Helgrind**:

```
==2608== For counts of detected and suppressed errors, rerun with: -v
==2608== Use --history-level=approx or =none to gain increased speed, at
==2608== the cost of reduced accuracy of conflicting-access information
==2608== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 172 from 27)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se puede observar que efectivamente la utilización de la memoria se hace de forma correcta.

Helgrind indica que no ha encontrado errores en cuanto a la programación multihilos. No es el resultado esperado puesto que se sabe que el programa finaliza sin que el hilo haya acabado su ejecución. Por lo tanto se deduce que Helgrind no es capaz de detectar este tipo de error y que no es útil para este caso.

8.2.9 Bug6.c

Este ejemplo muestra condiciones de carrera sobre una variable global, cuyo valor final es erróneo. Al enviarlo a la plataforma se obtiene lo siguiente:

Para **Memcheck**:

```
==3051== HEAP SUMMARY:
==3051==     in use at exit: 982 bytes in 5 blocks
```

```
==3051==   total heap usage: 15 allocs, 10 frees, 6,402,070 bytes allocated
==3051==
==3051== LEAK SUMMARY:
==3051==    definitely lost: 0 bytes in 0 blocks
==3051==    indirectly lost: 0 bytes in 0 blocks
==3051==      possibly lost: 0 bytes in 0 blocks
==3051==    still reachable: 982 bytes in 5 blocks
==3051==         suppressed: 0 bytes in 0 blocks
==3051== Rerun with --leak-check=full to see details of leaked memory
==3051==
==3051== For counts of detected and suppressed errors, rerun with: -v
==3051== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 15 from 8)
```

Para **Helgrind**:

```
==3426== Thread #3 was created
==3426==    at 0x4126728: clone (clone.S:111)
==3426==    by 0x40424B5: pthread_create@@GLIBC_2.1 (createthread.c:256)
==3426==    by 0x4026E2D: pthread_create_WRK (hg_intercepts.c:257)
==3426==    by 0x4026F8B: pthread_create@* (hg_intercepts.c:288)
==3426==    by 0x804878D: main (bug6.c:66)
==3426==
==3426== Thread #2 was created
==3426==    at 0x4126728: clone (clone.S:111)
==3426==    by 0x40424B5: pthread_create@@GLIBC_2.1 (createthread.c:256)
==3426==    by 0x4026E2D: pthread_create_WRK (hg_intercepts.c:257)
==3426==    by 0x4026F8B: pthread_create@* (hg_intercepts.c:288)
==3426==    by 0x804878D: main (bug6.c:66)
==3426==
==3426== Possible data race during read of size 4 at 0x804a03c by thread #3
==3426==    at 0x8048671: dotprod (bug6.c:37)
==3426==    by 0x4026F60: mythread_wrapper (hg_intercepts.c:221)
==3426==    by 0x4041E98: start_thread (pthread_create.c:304)
==3426==    by 0x412673D: clone (clone.S:130)
==3426==  This conflicts with a previous write of size 4 by thread #2
==3426==    at 0x8048679: dotprod (bug6.c:37)
==3426==    by 0x4026F60: mythread_wrapper (hg_intercepts.c:221)
==3426==    by 0x4041E98: start_thread (pthread_create.c:304)
==3426==    by 0x412673D: clone (clone.S:130)
==3426==
==3426== Possible data race during write of size 4 at 0x804a03c by thread #3
==3426==    at 0x8048679: dotprod (bug6.c:37)
==3426==    by 0x4026F60: mythread_wrapper (hg_intercepts.c:221)
==3426==    by 0x4041E98: start_thread (pthread_create.c:304)
==3426==    by 0x412673D: clone (clone.S:130)
==3426==  This conflicts with a previous write of size 4 by thread #2
==3426==    at 0x8048679: dotprod (bug6.c:37)
==3426==    by 0x4026F60: mythread_wrapper (hg_intercepts.c:221)
==3426==    by 0x4041E98: start_thread (pthread_create.c:304)
==3426==    by 0x412673D: clone (clone.S:130)
==3426==
==3426==
==3426== For counts of detected and suppressed errors, rerun with: -v
==3426== Use --history-level=approx or =none to gain increased speed, at
==3426== the cost of reduced accuracy of conflicting-access information
==3426== ERROR SUMMARY: 1392 errors from 2 contexts (suppressed: 1103
from 67)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se

puede observar que efectivamente la utilización de la memoria se hace de forma correcta.

Helgrind indica que se han encontrado dos situaciones de condición de carrera, es decir, que un hilo modifica una variable cuando el otro puede estar leyéndola, llegando a una situación final que posiblemente no era la esperada. Se observa así que Helgrind sí es capaz de detectar condiciones de carrera, lo cual es útil para el proyecto que se va a realizar.

8.2.10 Bug6fix.c

Este ejemplo muestra como se puede solucionar el problema de la condición de carrera utilizando una variable *mutex*. El resultado obtenido en la plataforma es el siguiente:

Para **Memcheck**:

```
==3070== HEAP SUMMARY:
==3070==     in use at exit: 982 bytes in 5 blocks
==3070==   total heap usage: 15 allocs, 10 frees, 642,070 bytes allocated
==3070==
==3070== LEAK SUMMARY:
==3070==    definitely lost: 0 bytes in 0 blocks
==3070==    indirectly lost: 0 bytes in 0 blocks
==3070==      possibly lost: 0 bytes in 0 blocks
==3070==    still reachable: 982 bytes in 5 blocks
==3070==         suppressed: 0 bytes in 0 blocks
==3070== Rerun with --leak-check=full to see details of leaked memory
==3070==
==3070== For counts of detected and suppressed errors, rerun with: -v
==3070== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 15 from 8)
```

Para **Helgrind**:

```
==3445== Thread #3 was created
==3445==    at 0x4126728: clone (clone.S:111)
==3445==    by 0x40424B5: pthread_create@@GLIBC_2.1 (createthread.c:256)
==3445==    by 0x4026E2D: pthread_create_WRK (hg_intercepts.c:257)
==3445==    by 0x4026F8B: pthread_create@* (hg_intercepts.c:288)
==3445==    by 0x80488B9: main (bug6fix.c:71)
==3445==
==3445== Thread #2 was created
==3445==    at 0x4126728: clone (clone.S:111)
==3445==    by 0x40424B5: pthread_create@@GLIBC_2.1 (createthread.c:256)
==3445==    by 0x4026E2D: pthread_create_WRK (hg_intercepts.c:257)
==3445==    by 0x4026F8B: pthread_create@* (hg_intercepts.c:288)
==3445==    by 0x80488B9: main (bug6fix.c:71)
==3445==
==3445== Possible data race during write of size 4 at 0x804a04c by thread #3
==3445==    at 0x8048785: dotprod (bug6fix.c:39)
==3445==    by 0x4026F60: mythread_wrapper (hg_intercepts.c:221)
==3445==    by 0x4041E98: start_thread (pthread_create.c:304)
==3445==    by 0x412673D: clone (clone.S:130)
==3445==  This conflicts with a previous read of size 4 by thread #2
==3445==    at 0x80487A2: dotprod (bug6fix.c:42)
```

```
==3445==    by 0x4026F60: mythread_wrapper (hg_intercepts.c:221)
==3445==    by 0x4041E98: start_thread (pthread_create.c:304)
==3445==    by 0x412673D: clone (clone.S:130)
==3445==
==3445==
==3445== For counts of detected and suppressed errors, rerun with: -v
==3445== Use --history-level=approx or =none to gain increased speed, at
==3445== the cost of reduced accuracy of conflicting-access information
==3445== ERROR SUMMARY: 70000 errors from 1 contexts (suppressed: 210720 from
43)
```

Análisis de resultados:

Memcheck indica que después del análisis no ha encontrado ningún tipo de error respecto al uso de la memoria. Viendo el código del programa en el apéndice D se puede observar que efectivamente la utilización de la memoria se hace de forma correcta.

Helgrind indica que ha encontrado una posible condición de carrera, aunque no era el resultado esperado puesto que este programa debería solucionar los problemas de sincronización del ejemplo anterior. Para este caso habría que revisar el código, puesto que se ha encontrado un error no esperado.

8.4 Resumen y conclusiones

Se han analizado con las herramientas Memcheck y Helgrind diez programas C que utilizan el estándar de programación concurrente POSIX pthreads. Cada uno tenía unas características concretas y se quería comprobar el resultado que ofrecían las herramientas de Valgrind en cada caso.

De todos los errores de estos diez programas, se ha comprobado que Memcheck y Helgrind detectan problemas de lectura/escritura ilegal y condiciones de carrera correspondientemente. El resto de problemas no han sido detectados, así que se ve aquí una de las limitaciones de la aplicación. Sin embargo, existen otras herramientas en Valgrind que podrían detectarlo, y eso sería una futura línea de trabajo.

Otro detalle a destacar es que todas las pruebas se han hecho a través de la plataforma implementada, accediendo a ella a través de la interfaz web y recibiendo la respuesta en el cliente de correo electrónico.

De la realización de las pruebas se llega a la conclusión de que la plataforma cumple con los objetivos especificados. Los módulos cumplen la función para la que han sido diseñados y el servidor de aplicaciones con la configuración empleada desempeña correctamente su trabajo.

Sin embargo, se observa que Valgrind puede no ofrecer los resultados que en principio se esperaban, concluyendo así que es necesario por parte del usuario un conocimiento amplio sobre la suite y qué posibilidades concretas ofrece.

Con el estado actual de desarrollo, el usuario puede obtener perfectamente el servicio para el que la plataforma ha sido diseñada.

Capítulo 9

Conclusiones y futuras líneas de trabajo

9.1 Conclusiones

El objetivo del proyecto era la creación de una plataforma JEE para ofrecer al usuario la posibilidad de validar sus programas en C remotamente, es decir, abstrayéndole de la necesidad de instalar en su equipo una herramienta extra. Para lograrlo se ha comenzado con el estudio de las tecnologías que iban a ser empleadas en el desarrollo del proyecto: Valgrind, Java EE y Glassfish.

Estudiando la suite de herramientas Valgrind se ha llegado a la conclusión de que es muy completa y ofrece más funcionalidades para el usuario que otras alternativas que también se habían valorado. Las funcionalidades más útiles son las de detección de problemas de memoria y de concurrencia, por eso en el acceso a la plataforma se dan facilidades para su uso. Sin embargo, la suite ofrece mucha más funcionalidad, por eso se ha optado por dar al usuario la opción de aprovechar al máximo la herramienta mediante un modo Experto.

En cuanto a la tecnología Java EE, se ha visto cómo facilita considerablemente la implantación de la plataforma. Esto es así porque hace que la aplicación desarrollada sea fácilmente portable a otros entornos. Además, ofrece la tecnología JMS, que aporta la ventaja mas notable proveyendo a la plataforma un sistema de encolado de peticiones que permite dar robustez a todo el proyecto. Por último, aporta unos mecanismos realmente útiles de cara a la presentación web (JSP) y recepción de peticiones (servlet).

Del análisis del servidor de aplicaciones Glassfish se ha comprobado que es muy completo en funcionalidad y sencillo de configurar. Es el más adecuado para el proyecto, y al ser de código libre, se puede obtener gratuitamente.

Después del diseño y el desarrollo se ha procedido a efectuar las pruebas de rendimiento correspondientes. De ellas se llega a la conclusión de que la plataforma funciona como se esperaba y ofrece la funcionalidad para la que ha sido diseñada. Tiene un comportamiento correcto incluso cuando aparecen errores que el usuario no contemplaba a priori, facilitando su corrección para conseguir al final el resultado deseado. Por todo esto, es una plataforma viable para el entorno docente que puede ser utilizada.

9.2 Futuras líneas de trabajo

Una futura línea de trabajo sería ofrecer en una misma plataforma la validación para Java y para C ensamblando el proyecto desarrollado por Manuel Jesús ([1]), como se explicó en el capítulo 2 de este documento. Así se crearía una aplicación muy completa.

Más localmente, en esta plataforma se podrían desarrollar más modos de uso que faciliten al usuario la utilización. Es decir, añadir a los modos Memchek y Helgrind otros que posibiliten el uso de las otras herramientas de Valgrind sin necesitar un conocimiento tan profundo como el que se requiere para el modo Experto. Además, como se indica en el apartado 8.4, habría que hacer un estudio y unas pruebas para concretar que tipos de problemas nuevos serían capaz de detectar estas herramientas.

Además, se podrían añadir módulos para extraer otros formatos de archivos comprimidos como Rar, Gzip o 7z. Debido a la modularidad de la plataforma, sería relativamente sencillo añadir otro módulo que se encargara de alguno de estos formatos.

En cuanto a la implantación real del servicio, podría ser útil para el administrador dotar a la plataforma de un sistema de autenticación. El servidor de aplicaciones ofrece algunas facilidades para tecnologías como LDAP.

Apéndices

Apéndice A.

Presupuesto

1: Autor: Pablo López Anastasio

2: Departamento: Ingeniería Telemática

3: Descripción del proyecto:

- **Título**: Diseño e implementación de aplicación web para validación remota de programas C
- **Duración (meses)**: 7
- **Tasa de costes indirectos**: 20%

4: Presupuesto total del Proyecto: **14.280 Euros**

5: Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	Categoría	Dedicación (hombres mes*)	Coste hombre mes	Coste (€)
Basanta Val, Pablo	Ingeniero SR	0,24	4.289,54	1.029,49
López Anastasio, Pablo	Ingeniero JR	4	2.694,39	10.777,56
			Total:	11.807,05

* 1 Hombre mes = 131,25 horas. Máximo anual de dedicación 12 hombres mes (1.575 horas). Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas).

EQUIPO

Descripción	Coste (€)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable**
PC con win XP y Office	800	100	7	60	93,33
				Total:	93,33

**Fórmula de cálculo de la amortización: $\frac{A}{B} xCxD$

A = nº de meses desde la fecha de la facturación en que el equipo es utilizado

B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total:	0

OTROS ASPECTOS DEL PROYECTO

Descripción	Empresa	Coste imputable
	Total:	0

6: Resumen de costes

Presupuesto Costes Totales	
Personal	11.807
Amortización	93
Subcontratación	0
Costes funcionamiento	0
Costes indirectos	2.380
Total:	14.280

Apéndice B.

Instalación y configuración de las herramientas utilizadas

B.1 Instalación JDK 6

Para instalar el Java Development Kit (JDK) 6 de Java ([3]) hay que seguir los siguientes pasos:

1. Ir a la página de descargas de Oracle:

   http://www.oracle.com/technetwork/java/javase/downloads/index.html

2. Obtener un instalador válido para el sistema operativo (Linux 32 bits en este caso) que contenga la versión deseada del JDK.

3. Ir hasta el directorio donde se ha descargado el instalador y ejecutar el siguiente comando para asegurarnos de que el ejecutable tiene los permisos necesarios:

   ```
   chmod a+x jdk-6u25-linux-i586.bin
   ```

4. Ir al directorio donde queremos instalar el JDK

   ```
   cd /usr/lib/jvm/
   ```

5. Ejecutar el archivo:

   ```
   sudo /home/usuario/Descargas/jdk-6u25-linux-i586.bin
   ```

B.2 Instalación de NetBeans

Para instalar NetBeans ([4]), que es el IDE utilizado en el proyecto que se va a realizar, hay que seguir los siguientes pasos:

1. Ir a la página de descargas de NetBeans:

http://netbeans.org/downloads/

2. Seleccionar idioma y sistema operativo (español y Linux en este caso).
3. Elegir la versión. En este caso es la que incluye Java EE y Glassfish.
4. Descargar
5. Ir hasta el directorio donde se ha descargado el instalador y ejecutar el siguiente comando para asegurarnos de que el ejecutable tiene los permisos necesarios:

```
chmod a+x archivodescargado.sh
```

6. Ir al directorio donde queremos instalar NetBeans
7. Ejecutar el archivo:

```
sudo sh /home/usuario/Descargas/archivodescargado
```

B.3 Instalación de Glassfish

No es necesario instalar ningún paquete nuevo, pues viene incluido en el IDE NetBeans que se describe anteriormente. A la hora de crear un proyecto, el propio entorno de desarrollo ofrece la posibilidad de elegir Glassfish como servidor de aplicaciones.

B.4 Instalación de Valgrind

Actualmente en los repositorios de Ubuntu se encuentra la versión 3.6.1, que es la utilizada en el proyecto que se va a realizar. Para instalarla basta con ejecutar el siguiente comando:

```
sudo apt-get install Valgrind
```

Para obtener la última versión hay que dirigirse a la sección de descargas de la página de Valgrind:

http://valgrind.org/downloads/current.html

Apéndice C.

Aspectos específicos de implementación

Esta sección del apéndice contiene los detalles del código de la implementación de algunos módulos o características interesantes del proyecto.

- Conexión JMS
- Extracción de archivos
- Compilación
- Ejecución
- Envío de correos

C.1 Conexión JMS

El API JMS o *Java Message Service* ([29]) es un estándar de mensajería que permite la aplicación basada en componentes en la plataforma *Java Enterprise Edition* (Java EE). Permite crear, enviar, recibir y leer mensajes.

JMS provee un servicio fiable y flexible para un intercambio asíncrono de información de negocio crítica. El API contiene los siguientes rasgos:

- Message-Driven Beans: permiten la recepción asíncrona de mensajes.
- Envíos y recepciones de mensajes JMS, que pueden participar en las transacciones de JTA (Java Transaction API).
- Interfaces de Java EE Connector Architecture que permiten integrar implementaciones JMS de otros proveedores.

La inclusión del API de JMS mejora la plataforma Java EE simplificando el desarrollo de las aplicaciones. Permite interacciones entre componentes de dicha plataforma y sistemas capaces de enviar mensajes, de forma débilmente acoplada, fiable y asíncrona.

La arquitectura del contenedor de los Enterprise JavaBeans (EJBs) de la plataforma J2EE mejora el API de JMS de dos formas:

- Permitiendo el consumo concurrente de mensajes.
- Proveyendo soporte para transacciones distribuidas como actualización de bases de datos, procesamiento de mensajes y conexiones con sistemas EIS utilizando la Java EE *Connector Architecture*, pudiendo participar todos en el mismo contexto.

El funcionamiento que se desea obtener de esta tecnología es éste: Una aplicación cliente envía mensajes a la cola (queue), el proveedor de JMS (en este caso el Servidor Java EE) entrega los mensajes a las instancias de Message-Driven Beans (MDB), las cuales procesarán los mensajes.

C.1.2 Implementación

El módulo de JMS utilizado en el proyecto se compone de dos partes:

- Cliente: En el servlet de recepción de peticiones se usa para enviar las peticiones a la cola de JMS.
- Message-Driven Bean: Que se encarga de gestionar los mensajes de la cola.

C.1.2.1 Creación de cliente

Para la creación de clientes se necesitará utilizar las siguientes clases:

- *javax.naming.InitialContext*
- *javax.jms.QueueConnectionFactory*
- *javax.jms.Queue, javax.jms.QueueConnection*
- *javax.jms.QueueSession*
- *javax.jms.QueueSender*

Primero se crearán atributos en la clase cliente. Estos atributos son de las clases anteriormente indicadas:

```
InitialContext jndiContext = null;
QueueConnectionFactory queueConnectionFactory = null;
Queue queue = null;
QueueConnection conexion = null;
QueueSession sesion = null;
QueueSender mensajero = null;
```

Después se inicializarán. El atributo `jndiContext` de la clase `InitialContext` se utilizará para obtener el contexto inicial del directorio de nombres (JNDI) y se inicializa llamando al constructor por defecto:

```
jndiContext = new InitialContext();
```

Seguidamente se deberán obtener `QueueConnectionFactory` y `Queue`. `QueueConnectionFactory` modela las conexiones con la cola (`Queue`) de mensajes y se obtiene utilizando `jndiContext` de esta forma:

```
queueConnectionFactory = (QueueConnectionFactory)
jndiContext.lookup("jms/MyQueueConnectionFactory");
```

Se utiliza el método `lookup()`, el cual busca en el directorio de nombres el recurso, que en este caso se llama *jms/MyQueueConnectionFactory*, y obtiene la instancia. No es un nombre al azar, ya que debe haberse definido en el servidor de aplicaciones y registrado con el nombre indicado para que funcione. Para la cola de mensajes (`Queue`) se utilizará el mismo método:

```
queue = (Queue) jndiContext.lookup("jms/MyQueue");
```

Tanto la inicialización de `InitialContext` como las de `QueueConnectionFactory` y `Queue` pueden lanzar una excepción `NamingException`.

Para crear la conexión se utiliza la instancia de `QueueConnectionFactory` y con ella se llama a su método `createQueueConnection()`. Este método genera una instancia de la clase `QueueConnection` que se puede almacenar en la variable `conexion` inicializada con valor `null`.

```
conexion = queueConnectionFactory.createQueueConnection();
```

Se puede iniciar una sesión con la cola a través de esta conexión utilizando el método `createQueueSession(boolean transacted, int acknowledgeMode)` de la clase `QueueConnection`. El primer parámetro indica si el envío es con confirmación de recepción (`false`) o no (`true`). El segundo, indica el tipo de confirmación y será ignorado en caso de que el primer parámetro tenga valor `true`. Tipos de confirmación:

- **`Session.AUTO_ACKNOWLEDGE`**: asentimiento automático, cuando se recibe bien un mensaje de un cliente.
- **`Session.CLIENT_ACKNOWLEDGE`**: asentimiento de cliente, el cual llama al método de asentimiento del mensaje. Confirmando el mensaje del que se ha llamado el método de asentimiento, se confirman todos los mensajes que la sesión ha consumido.
- **`Session.DUPS_OK_ACKNOWLEDGE`**: asentimiento que permite duplicados. Indica a la sesión que confirme vagamente la entrega de mensajes. Los consumidores deben soportar mensajes duplicados.

La instancia obtenida se almacena en el atributo de la clase `QueueSession` (sesion) de la siguiente manera (utilizando, en este caso, una sesión con confirmación automática, que suele ser lo más común):

```
sesion = conexion.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
```

Para terminar de inicializar los elementos necesarios se obtiene una instancia de la clase `QueueSender` (variable `mensajero`). Dicha instancia se obtiene mediante el método de la clase `QueueSession`, `createSender (Queue queue)`. Se le pasa por parámetro la instancia de la clase `Queue` (variable `queue`):

```
mensajero = sesion.createSender(queue);
```

La instanciación de `QueueConnectionFactory`, `QueueSession` y `QueueSender` puede lanzar una excepción del tipo `JMSException` que deberá ser capturada adecuadamente.

Finalmente, para terminar el código concerniente al cliente resta la creación y envío de un mensaje. Los mensajes son modelados por la interfaz `Message` (también del paquete `javax.jms`) y contiene los métodos necesarios para construir, añadir propiedades o llamar a confirmación. Algunos métodos de esta interfaz que pueden ser de utilidad:

- **`void acknowledge()`:** confirma el mensaje con el que se llama al método y todos los anteriores (relacionado con el tipo de confirmación en modo cliente).
- **`void clearBody()`:** Borra el cuerpo del mensaje.
- **`get/setJMSDeliveryMode()`:** Obtiene o configura el modo de entrega (si es `set`, se le pasa un tipo `int` indicando el modo).
- **`get/setJMSDestination()`:** Obtiene o configura el destino (si es `set`, se le pasa una instancia de tipo `Destination`).
- **`get/setJMSTimestamp()`:** Obtiene o configura la marca de tiempo del mensaje (si es `set`, se indica con un tipo `long`).

Estos métodos pueden usarse tanto en envío, como en recepción.

Para una aplicación sencilla, la clase que puede modelar nuestro mensaje es una sub-interfaz llamada `TextMessage`. Los métodos que añade a de los que debe implementar por herencia son: `void setText(java.lang.String)` y `java.lang.String getText()`. En código:

```
TextMessage message = null;
message = sesion.createTextMessage();
message.setText(".............");
```

Se utiliza el método `createTextMessage()`, de la clase `QueueSession`, para generar la instancia de `TextMessage`. Una vez que se ha generado el mensaje con el contenido deseado, se procede al envío utilizando la instancia de `QueueSender` (variable `mensajero`). Para ello se utiliza el método `send(Message message)` de la clase `QueueSender`:

```
mensajero.send(message);
```

C.1.2.2 Creación de un Message-Driven Bean

Para modelar un Message-Driven Bean (EJB 3.0) se pueden utilizar diferentes IDEs. En este caso se ha utilizado NetBeans, que ofrece un asistente completo con el que es muy sencillo crear un MDB.

Una vez creado se completa la anotación @MessageDriven que aparece justo antes de la declaración de la clase, de esta manera:

```
@MessageDriven (activationConfig = { @ActivationConfigProperty(
      propertyName = "destinationType",
      propertyValue = "javax.jms.Queue")},
      mappedName = "jms/MyQueue",
      messageListenerInterface = MessageListener.class)
```

Mediante ésta anotación de los EJB 3.0, se indica qué cola de mensajes debe escuchar el MDB.

El parámetro `activationConfig` indica la configuración del MDB en su entorno de operación. Esto incluye información sobre modos de asentimiento, destinos esperados o tipos de puntos finales (*endpoints*), entre otros. La configuración de éste parámetro se especifica usando un array de anotación del tipo `@javax.ejb.ActivationConfigProperty`, el cual especifica las propiedades mediante nombre (`propertyName`) y valor (`propertyValue`). En este caso, se indica que la propiedad a configurar es el tipo de destino y el valor de dicho destino es un objeto de la clase `javax.jms.Queue`.

El parámetro `mappedName` se utiliza para indicar el nombre JNDI del destino de mensajes del MDB. El tipo de datos que se introduce es un `String`, que en este caso toma el valor `jms/MyQueue`. Éste debe coincidir con el nombre de la cola obtenido del directorio de nombres en el cliente.

El parámetro `messageListenerInterface` indica la interfaz escuchadora de mensajes que utiliza MDB. Se debe utilizar si el MDB no implementa explícitamente la interfaz mensaje-escuchador, o si implementa más de una interfaz del tipo `java.io.Serializable`, `java.io.Externalizable` o cualquiera de las interfaces del paquete `javax.ejb`. El tipo de datos es `Object.class`.

Otros atributos de ésta anotación, no utilizados en este caso, son:

- `name`: Especifica el nombre del MDB. Por defecto es el nombre no cualificado de la clase (*unqualified name*).
- `description`: Especifica una descripción del MDB.

Ninguno de los cinco atributos indicados es obligatorio.

El MDB debe implementar las interfaces `MessageListener` y `MessageDrivenBean`:

```
public class JMSBean implements MessageListener,
MessageDrivenBean {
```

Una vez hecho esto, se debe implementar el método de la interfaz `MessageListener` que manejará la recepción de los mensajes. Se le llamará automáticamente (cuando la aplicación esté desplegada en el servidor) al recibir en la cola un mensaje. Dentro de éste método se pueden utilizar los métodos de la clase `Message` y `TextMessage` para obtener la información necesaria.

```
public void onMessage(Message inMessage)
```

Un ejemplo del contenido del método onMessage es el que sigue:

```
TextMessage msg = null;
String datos = null;
try{
      //Si lo recibido es un mensaje de texto...
      if(inMessage instanceof TextMessage){
            //Se almacena
            msg = (TextMessage) inMessage;
            //Se pasa a String
            datos = msg.getText();
            System.out.println("MESSAGE BEAN: Mensaje recibido:
            "+msg.getText());
      }else{
            System.out.println("Mensaje de tipo erroneo:
            "+inMessage.getClass().getName());
      }
}catch(JMSException e){
      e.printStackTrace();
}catch(Throwable te){
      te.printStackTrace();
}
```

Otros métodos que pueden aparecer al implementar `MessageDrivenBean` son:

```
@Override
public void ejbRemove()throws EJBException {}

@Override
public     void     setMessageDrivenContext(MessageDrivenContext
arg0)throws EJBException {}
```

Se pueden dejar vacíos, ya que pertenecen al ciclo de vida y contexto del MDB. De esta forma se deja que el contenedor de EJBs del servidor lo maneje a su antojo.

C.1.2.3 Configuración de los recursos JMS en el servidor Glassfish

Primero se debe entrar en la consola de administración del servidor Glassfish (versión utilizada 3.1.1). Normalmente se puede acceder introduciendo en cualquier navegador web la dirección http://localhost:4848 (se utiliza *localhost* si el servidor está instalado en la máquina local, si no se utiliza la dirección IP).

Una vez introducidos usuario y contraseña (se necesitan permisos de administrador), en la lista de la izquierda de la pantalla se selecciona *JMS Resources*. Está ubicado dentro de la pestaña *Resources*, como se muestra en la figura 40.

Figura 40: JMS Resources en Glassfish

Para configurar una instancia del tipo `QueueConnectionFactory` se selecciona la carpeta *Connection Factories*. Aparece una pantalla a la derecha de esta lista, que se puede ver en la Figura 41, en la que se pulsa el botón new.

Figura 41: Pantalla JMS Connection Factories

Una vez hecho esto, aparece otra pantalla con un formulario que en este caso se rellena como se indica en la Figura 42.

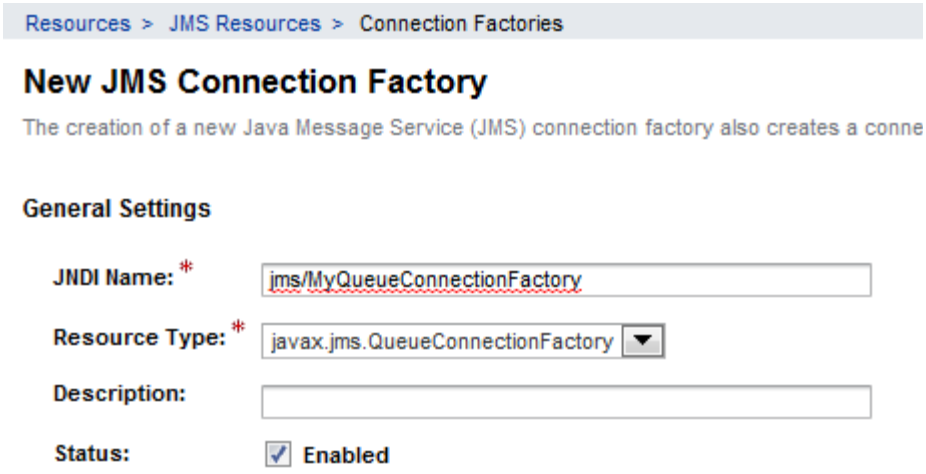

Figura 42: Formulario de creación de QueueConnectionFactory

En la Figura 42 se puede ver que se indica el nombre JNDI y el tipo de recurso. En este caso es *jms/MyQueueConnectionFactory* y *javax.jms.QueueConnectionFactory* respectivamente, ya que es el nombre del tipo `QueueConnectionFactory` que se utiliza en el ejemplo de la creación del cliente JMS y del MDB. Debe ser el mismo para que funcione.

Después se indica si las transacciones JMS serán locales o remotas de la forma que se muestra en la Figura 43 (en este caso locales).

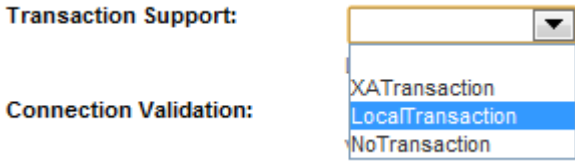

Figura 43: Tipo de transacciones JMS

Seguidamente se guardan los datos introducidos mediante un botón *OK* que se encuentra a la derecha de lo mostrado en la Figura 48.

Para configurar una instancia de `Queue` se pulsa la pestaña *Destination Resources*, debajo de *JMS Resources*. Aparece una pantalla a la derecha, mostrada en la Figura 44, en la cual, se pulsa el botón *new*.

Figura 44: Pantalla Destination Resources

Esto muestra un formulario, mostrado en la Figura 45, en el que se puede introducir un nombre JNDI, en nuestro caso *jms/MyQueue*, un nombre de destino físico, en el que se introduce el nombre cualquier nombre, en nuestro caso *JMSBean* y el tipo de recurso, que será *javax.jms.Queue*. El nombre *JMSBean* hay que recordarlo pues se utiliza en otro formulario de configuración.

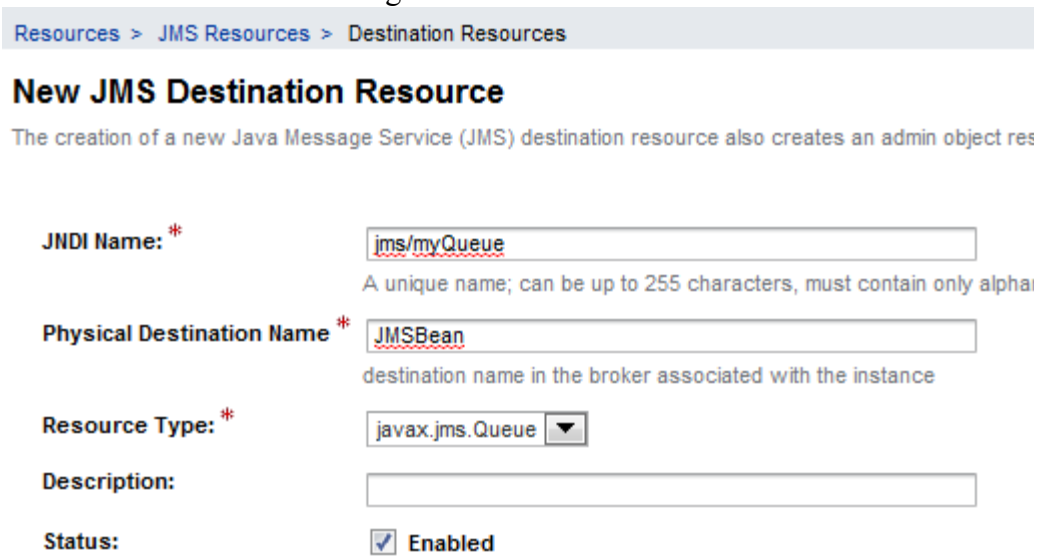

Figura 45: Formulario de creación de Queue

Seguidamente se guardan los datos introducidos mediante un botón *OK* que se encuentra a la derecha de lo mostrado en la Figura 40.

Se procede a pulsar en una pestaña llamada *Java Message Service*, dentro de la pestaña *Configuration*, tal y como se muestra en la figura 46.

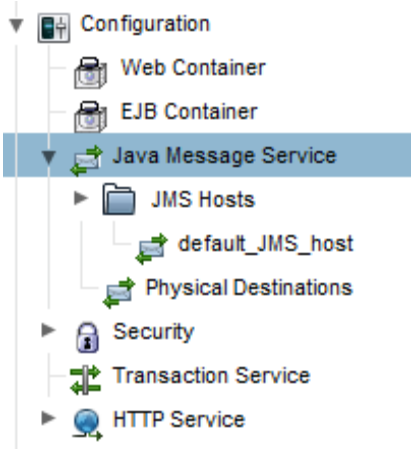

Figura 46: Pestaña de configuración de JMS

Aparece una pantalla con opciones configurables, en las que en este caso, hay que introducir los datos que se ven en la Figura 47.

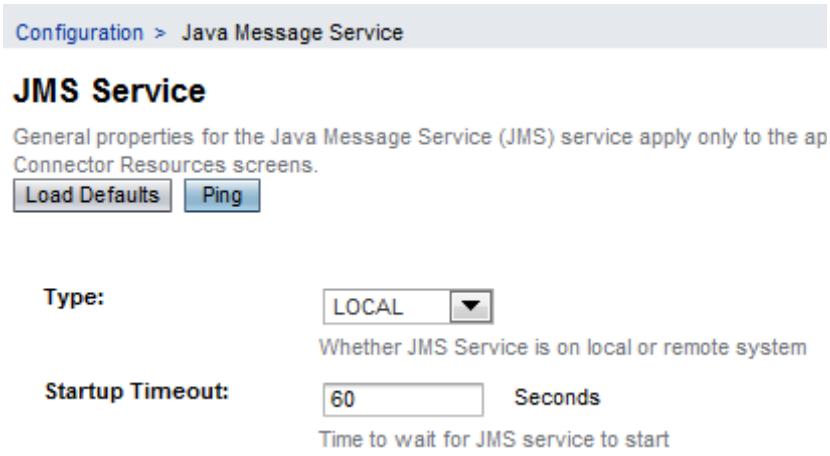

Figura 47: Configuración del servicio JMS del servidor

Después de guardar los datos anteriores, se pulsa en la pestaña *Physical Destinations*, aparece lo mostrado en la Figura 48 y se pulsa el botón *new* de la pantalla que aparece en el lado derecho.

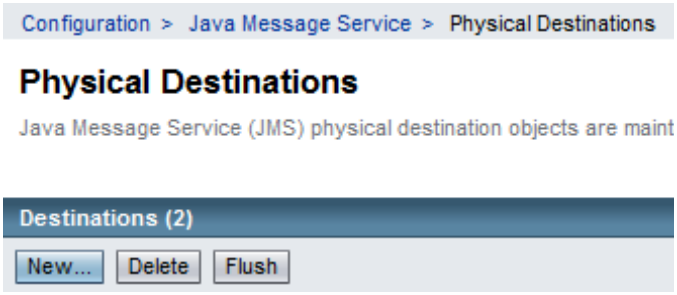

Figura 48: Pantalla de Physical Destinations

Se muestra un formulario que hay que rellenar de la forma que se indica en la Figura 49.

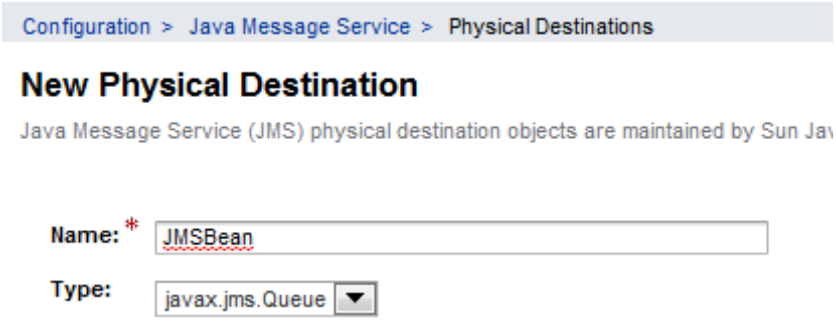

Figura 49: Formulario de Physical Destination

En este formulario es en el que hay que introducir el nombre del destino físico utilizado al crear la cola *JMSBean*. En el tipo de destino físico se indica *javax.jms.Queue*.

Para finalizar con la configuración se accede a la pestaña *EJB Container*, como se muestra en la Figura 50.

Figura 50: Pestaña EJB Container

El lado derecho de la pantalla entonces muestra tres pestañas. Se selecciona *MDB settings* y se muestra el formulario de configuración de la Figura 51.

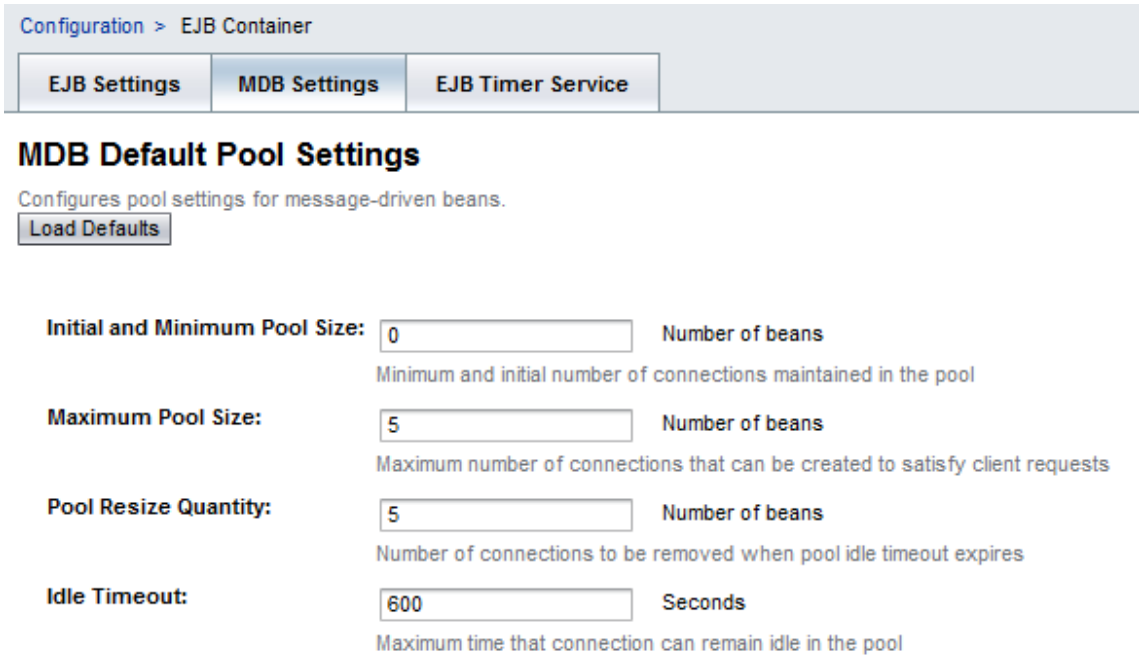

Figura 51: Opciones de MDB en Glassfish

En este formulario se permite configurar, entre otras propiedades, el número máximo de peticiones de la cola de mensajes que gestionará a la vez el MDB (opción *Maximum Pool Size*). Una vez configurado esto, ya debe funcionar la conexión JMS.

C.2 Extracción de archivos

Para la extracción de archivos comprimidos se ha utilizado la librería de java *java.util.zip* ([16])

Primero se inicializan las instancias a utilizar en la extracción:

```
File archivo= new File (“Archivo.zip”);
String directorio =”Directorio_de_extracción”;
ZipFile azip = new ZipFile (archivo);
```

La instanciación de `ZipFile` puede lanzar una excepción del tipo `IOException`. Se crean los flujos que permiten la lectura del archivo comprimido:

```
FileInputStream fis = null;
BufferedInputStream bis = null;
```

Se incializaría de esta forma:

```
fis = new FileInputStream(azip.getName());
bis = new BufferedInputStream(fis);
ZipInputStream jis = new ZipInputStream(bis);
```

Para ir obteniendo archivo a archivo de la instancia de `ZipInputStream` se necesita un bucle que vaya leyendo de este flujo y almacenarlo en alguna variable. Para leer del flujo existe este método:

`ZipInputStream.getNextEntry()`, que devuelve una instancia del tipo `ZipEntry`.

Para escribir lo obtenido en un archivo en el directorio deseado, primero se debe crear un flujo del tipo `FileOutputStream` (variable *fos*), al que se le pasa un tipo *String* con la dirección completa del archivo de salida (utilizando la variable *directorio* inicializada al principio). Después se crea un *buffer* de escritura de datos de archivo. Éste debe ser del tipo `BufferedOutputStream`, al que se le debe pasar como parámetro el flujo mencionado y un entero que indique el tamaño del *buffer* (en este caso, 8192 es suficiente). Mediante un bucle se va leyendo los datos de la entrada, con un método de la clase `ZipInputStream` , y escribiendo en el nuevo archivo con un método de `BufferedOutputStream`:

```
int count =0;
BufferedOutputStream dest = new BufferedOutputStream(fos, 8192);
while((count = jis.read(data, 0, 8192)) != -1 ){
dest.write( data, 0, count );
}
```

La variable *data* es un *array* de tipo byte del mismo tamaño que el buffer de escritura de datos (8192).

El bucle completo es de esta forma:

```
int count = 0;
byte data[] = new byte[8192];
ZipEntry/JarEntry entry;

// Se van obteniendo las entradas del flujo
while((entry= jis.getNextEntry()/getMextJarEntry())!= null ){

      //Se genera el nombre del archivo con su dirección completa
      //Aquí se incluye el directorio de extracción
      String destFN = directorio + File.separator +
      entry.getName();

      //Se genera un formato correcto de dirección del archivo
      if( destFN.indexOf("/") != -1 ){
            destFN = destFN.replace("/", File.separator);
      }
```

```
            //Si la entrada no es un directorio..
            if( !entry.isDirectory()){

                  //Se guarda el archivo en la unidad
                  //Primero se crea el flujo para la escritura
                  FileOutputStream fos = null;
                  try{

                        fos = new FileOutputStream( destFN );

                  }catch(FileNotFoundException fnf){

                        // En caso de que haya alguna carpeta
                        // intermedia que no se haya generado, se crea
                        int endIndex =
                        destFN.lastIndexOf(File.separator);
                        String aux = destFN.substring(0, endIndex);
                        File auxf = new File (aux);
                        auxf.mkdirs();
                        // Y se crea el flujo
                        fos = new FileOutputStream( destFN );
                  }

                  //Se escribe el contenido del archivo el contenido
                  // del archivo
                  dest = new BufferedOutputStream( fos, 8192);
                  while((count = jis.read(data, 0, 8192)) != -1 ){
                        dest.write( data, 0, count );
                  }

                  //Se cierran los flujos de escritura de archivo
                  dest.flush();
                  dest.close();
                  fos.flush();
                  fos.close();

            // Si es un directorio, se crea
            }else{
                  File f = new File(destFN);
                  f.mkdirs();
            }
      }

      //Se cierran el resto de flujos
      jis.closeEntry();
      jis.close();
      fis.close();
      azip.close();

      //Fin de extracción
```

Esto es todo lo necesario para extraer los archivos comprimidos que se utilizan en el proyecto.

C.3 Compilación

Para llevar a cabo la compilación, el módulo hace una llamada al sistema operativo desde el propio código Java para ejecutar un comando como si estuviéramos en la ventana de comandos. Ese comando está compuesto por el compilador gcc, los ficheros .c a compilar y las opciones. Cada componente deberá ser un *string*, para luego formar un array de *strings* que se le pasará al ejecutor de comandos. Ahora vamos a ver cómo se obtienen las distintas partes.

Como se ve en la figura 25 (apartado 6.6), lo primero que se hace es listar los ficheros con extensión .c que se encuentran en el directorio */Peticiones/ID_rdm/ext*. Para ello se utiliza la librería *java.util.LinkedList* ([30]) y una clase llamada `BuscadorFicheros` que tiene un método `dameFicheros()` encargado de examinar el directorio que se le pasa por parámetro y aplicar una mascara de comparación para obtener una lista de los ficheros que cumplen esa máscara.

```
LinkedList<File> ficherosC = new LinkedList<File>();
BuscadorFicheros.dameFicheros(dir_ext, "*.c", ficherosC, true);
```

Después hay que analizar las opciones de compilación que ha introducido el usuario. Éstas vienen codificadas en un *string* cuyo formato de es el siguiente:

```
-opcion1 -opcion2 -opcion3
```

Para separar las distintas opciones se utiliza la librería *java.util.StringTokenizer* ([31]). Se crea un objeto de la clase `StringTokenizer` pasándole al constructor el string que queremos decodificar.

```
StringTokenizer tokens = new StringTokenizer(compilacion);
```

Luego mediante el método `nextToken()` vamos obteniendo los substrings y guardándolos en un array con todas las opciones.

```
int N_opciones = 0;
String opt = "";
String opciones[] = new String [10];
while(tokens.hasMoreTokens()){
      opt = tokens.nextToken();
      opciones[N_opciones] = opt;
      N_opciones++;
}
```

Ya tenemos todo lo necesario y ahora hay que crear el array de strings con los elementos en el orden correcto. El comando quedará de la siguiente forma:

```
gcc –g fich1.c fich2.c fichN.c –o ejecutable –opc1 –opc2 -opcN
```

Si todo va bien, el resultado siempre será un fichero ejecutable con el nombre "ejecutable" guardado en el directorio */Peticiones/ID_rdm/bin* independientemente de las opciones del usuario.

Para ejecutar el comando en cuestión se utiliza el siguiente código, donde `comando` es el array de strings creado antes:

```
final Process proc = Runtime.getRuntime().exec(comando);
InputStream is = proc.getErrorStream();
int size;
String s;
int exCode = proc.waitFor();
StringBuffer ret = new StringBuffer();
while((size = is.available()) != 0) {
      byte[] b = new byte[size];
      is.read(b);
      s = new String(b);
      ret.append(s);
}
```

Si se produce algún error en la compilación, se procede a recoger el `ErrorStream` y se vuelca en un fichero de texto que será el *log* de error de compilación que posteriormente será enviado al usuario.

```
if(ret.length()>0){
      PrintWriter pw = null;
      fichero = new FileWriter(dir_bin+"/CompilacionLog.txt");
      pw = new PrintWriter(fichero);
      pw.println(ret.toString());
}
```

C.4 Ejecución

Para la ejecución el proceso es muy similar al de compilación (apartado anterior). Hay que crear un array de strings formando el comando a ejecutar por `Runtime.getRuntime().exec(comando)`.

Si el modo elegido por el usuario es Memcheck o Helgrind, el comando a ejecutar será uno de los dos siguientes:

```
valgrind –tool=memcheck ejecutable
valgrind –tool=helgrind ejecutable
```

Sin embargo, si el modo es Experto, habrá que analizar el string `comando` que fue introducido por el usuario en el formulario inicial. La forma de hacerlo es, igual que para las opciones de compilación, con la librería *StringTokenizer*.

```
StringTokenizer tokens = new StringTokenizer(comando);
int n = 0;
String t = "";
while(tokens.hasMoreTokens()){
      t = tokens.nextToken();
```

```
        if (t.equals("MiPrograma")){
              command[n] = dir_bin+"/ejecutable";
        }else{
              command[n] = t;
        }
        n++;
  }
```

Como se puede ver en el fragmento de código anterior, el substring `MiPrograma` es sustituido por la dirección del ejecutable creado en la fase de compilación. Así se abstrae al usuario del control del árbol de directorios. Además, se comprueba, por seguridad, que el primer elemento se corresponde con la ejecución de Valgrind. Así el usuario no puede ejecutar otro programa en el servidor.

```
if (command[0].equals("valgrind")){
      ...
else {//SOLO SE PUEDE EJECUTAR VALGRIND
      return = -2;
}
```

Después, igual que en el módulo de compilación, se procede a ejecutar el comando sintetizado. La diferencia es que el `ErrorStream` se lee y se vuelca en un fichero siempre, puesto que su contenido es la información que Valgrind nos devuelve. Dicho fichero será `ValgrindLog.txt` y su contenido se enviará por email al usuario.

Sin embargo, también pueden ocurrir varios tipos de errores en el proceso. Por eso, el módulo Ejecutor devuelve el resultado al módulo Validador en forma de número entero. Los posibles valores y su significado son los siguientes:

- 0 = Ejecución exitosa.
- -1 = Tiempo de ejecución excedido.
- -2 = Error en la ejecución.

Si ocurriera cualquiera de los dos errores, el usuario sería informado.

C.4.1 Temporizador

Podemos encontrarnos habitualmente con programas que, debido a errores en su código, nunca acaban su ejecución. Si no se atendiera a esta problemática, el servidor acabaría siempre saturado con un número de programas ejecutándose infinitamente.

Para evitar ese problema, se ha optado por implementar un temporizador que controle el tiempo de ejecución de los programas de las peticiones de los usuarios. Se ha hecho utilizando las APIs *javax.swing.Timer* ([32]) y *java.awt.event*([33]).

Primero se crea el temporizador, definiendo el tiempo en milisegundos que queremos que transcurra hasta que salte el evento, y especificando que acciones queremos que se lleven a cabo cuando esto ocurra.

```
milisegundos = 60000*15;
Timer timer = new Timer (milisegundos, new ActionListener ()
{

      public void actionPerformed(ActionEvent e)
      {

      ejecucion_ok = -1; //fuera de tiempo
      proc.destroy();
      System.out.println("tiempo! Ya han pasado los
                        "+milisegundos/60000+" minutos.");


}
});
```

En cuanto al tiempo máximo, será decisión del administrador definir cuál será. Dependerá del tipo de aplicaciones que espera recibir.

El temporizador será iniciado con el método `start()` justo antes de comenzar la ejecución, y será parado con el método `stop()` justo después, en el caso de que la ejecución llegue a su fin. De no ser así, saltaría el evento, que, como hemos visto anteriormente, finalizaría el proceso y devolvería un valor -1 indicando que se ha excedido el tiempo.

C.5 Envío de correos

En el proyecto se necesita enviar un correo al terminar la validación. Para este fin se utiliza la librería *javax.mail* ([15]). Para enviar un correo se necesita una cuenta de correo, y un servidor de correo que lo entregue.

Para solucionar esto, en este caso, se ha utilizado Gmail. Se ha creado una cuenta de correo para nuestra aplicación, cuya dirección es:

cvalidacion.uc3m@gmail.com.

De esta forma, teniendo una cuenta Gmail el servidor permitirá enviar correos a cualquier dirección.

Para enviar un correo, lo primero es configurar el envío para que conecte con el servidor de Gmail. Para esto se crea una instancia de la clase `java.util.Properties`:

```
java.util.Properties props = new java.util.Properties();
```

Para ir añadiendo propiedades se utilice el método `put()` de la clase `Properties`, al que se le pasan 2 parámetros:

- `String propiedad`: Indica el nombre de la propiedad a configurar.
- `String valor`: Indica el valor de la propiedad a configurar.

Para la correcta configuración de la conexión con Gmail, éstas son todas las propiedades necesarias:

```
props.put("mail.smtp.host", "smtp.gmail.com");
props.put("mail.smtp.port", "587");
props.setProperty("mail.smtp.starttls.enable", "true");
props.setProperty("mail.smtp.user","javapathfinder.uc3m@gmail.com");
props.setProperty("mail.smtp.auth", "true");
```

Con estas propiedades se configura una sesión, es decir, se crea una instancia de la clase `javax.mail.Session` utilizando el método estático `getDefaultInstance()` de `Session`:

```
Session session = Session.getDefaultInstance(props, null);
```

Se le pasa como parámetros el objeto que contiene las propiedades y una instancia de la clase `Authenticator` que en este caso no es necesaria, por lo que se pasa un valor `null`.

Después se construye el mensaje. Se crea una instancia de la clase `javax.mail.Message` utilizando el constructor `MimeMessage()` al que se le pasa la instancia que contiene la sesión.

```
Message msg = new MimeMessage(session);
```

Seguidamente, se van incluyendo en el mensaje los campos necesarios para el envío de esta forma:

```
msg.setFrom(new InternetAddress("JPFPathfinder@it.uc3m.es"));
msg.setRecipient(Message.RecipientType.TO, new
                  InternetAddress(“MAIL_DE_DESTINO”));
msg.setSubject(“ASUNTO”);
msg.setText(“CONTENIDO”);
```

Para el envío de este mensaje se crea una instancia del tipo `javax.mail.Transport` mediante el método `getTransport()` de la clase `Session` que recibe como parámetro el tipo de protocolo de transporte, que en este caso es smtp.

```
Transport t = session.getTransport("smtp");
```

Utilizando los datos de acceso a nuestra cuenta de correo, se lanza una conexión con el método `connect()` de la clase `Transport`.

```
t.connect("javapathfinder.uc3m@gmail.com","jpf12345");
```

Finalmente se llama al método `sendMessage()` de la clase `Transport`, con parámetros:

- `Message mensaje`: la instancia de la clase `Message` generada.
- `Address[] direcciones`: array con las direcciones de destino del mensaje. Se puede obtener mediante el método `getAllRecipients()` de la clase `Message`.

```
t.sendMessage(msg,msg.getAllRecipients());
```

Y se cierra la conexión:

```
t.close();
```

Apéndice D.

Aplicaciones de prueba

En esta sección se encuentra el código de las aplicaciones C que se han empleado para realizar las pruebas detalladas en el Capítulo 7. Se han obtenido del enlace [28].

Bug1.c

```
/*****************************************************************
* FILE: bug1.c
* DESCRIPTION:
*     This example has a bug. It is a variation on the condvar.c
example.
*   Instead of just one thread waiting for the condition signal, there
are
*   four threads waiting for the same signal. Find out how to fix the
*   program. The solution program is bug1fix.c.
*  SOURCE: Adapted from example code in "Pthreads Programming", B.
Nichols
*   et al. O'Reilly and Associates.
* LAST REVISED: 07/06/05  Blaise Barney
*****************************************************************/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  6
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
  int j,i;
  double result=0.0;
  long my_id = (long)idp;
  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
```

```
    Check the value of count and signal waiting thread when condition
is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count():  thread  %ld,  count  =  %d     Threshold
reached.\n", my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *idp)
{
  long my_id = (long)idp;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
  Lock  mutex  and  wait  for  signal.   Note  that  the  pthread_cond_wait
routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run
by
  the  waiting  thread,  the  loop  will  be  skipped  to  prevent
pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
    printf("***Before cond_wait: thread %ld\n", my_id);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("***Thread %ld Condition signal received.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  pthread_t threads[6];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /*
  For portability, explicitly create threads in a joinable state
  */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[2], &attr, watch_count, (void *)2);
  pthread_create(&threads[3], &attr, watch_count, (void *)3);
  pthread_create(&threads[4], &attr, watch_count, (void *)4);
```

```
  pthread_create(&threads[5], &attr, watch_count, (void *)5);
  pthread_create(&threads[0], &attr, inc_count, (void *)0);
  pthread_create(&threads[1], &attr, inc_count, (void *)1);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);

}
```

Bug1fix.c

```
/******************************************************************
* FILE: bug1fix.c
* DESCRIPTION:
*   Solution for the bug1.c program.  The inc_count routine uses a
*   pthread_cond_broadcast() routine instead of the
pthread_cond_signal()
*   routine.
* SOURCE: Adapted from example code in "Pthreads Programming", B.
Nichols
*   et al. O'Reilly and Associates.
* LAST REVISED: 01/29/09  Blaise Barney
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  6
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
  int j,i;
  double result=0.0;
  long my_id = (long)idp;
  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /* Check the value of count and signal waiting thread when
condition is reached.  Note that this occurs while mutex is locked. */
    if (count == COUNT_LIMIT) {
      pthread_cond_broadcast(&count_threshold_cv);
```

```
      printf("inc_count(): thread %ld, count = %d  Threshold
reached.\n",
             my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
         my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);

    }
  pthread_exit(NULL);
}

void *watch_count(void *idp)
{
  long my_id = (long)idp;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /* Lock mutex and wait for signal.  Note that the pthread_cond_wait
Soutine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run
by the waiting thread, the loop will be skipped to prevent
pthread_cond_wait from never returning.
  */
  pthread_mutex_lock(&count_mutex);
    printf("***Before cond_wait: thread %ld\n", my_id);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("***Thread %ld Condition signal received.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
int i, rc;
pthread_t threads[6];
pthread_attr_t attr;

/* Initialize mutex and condition variable objects */
pthread_mutex_init(&count_mutex, NULL);
pthread_cond_init (&count_threshold_cv, NULL);

/* For portability, explicitly create threads in a joinable state */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
pthread_create(&threads[2], &attr, watch_count, (void *)2);
pthread_create(&threads[3], &attr, watch_count, (void *)3);
pthread_create(&threads[4], &attr, watch_count, (void *)4);
pthread_create(&threads[5], &attr, watch_count, (void *)5);
pthread_create(&threads[0], &attr, inc_count, (void *)0);
pthread_create(&threads[1], &attr, inc_count, (void *)1);

/* Wait for all threads to complete */
for (i = 0; i < NUM_THREADS; i++) {
  pthread_join(threads[i], NULL);
  }
printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);
```

```
/* Clean up and exit */
pthread_attr_destroy(&attr);
pthread_mutex_destroy(&count_mutex);
pthread_cond_destroy(&count_threshold_cv);
pthread_exit (NULL);

}
```

Bug2.c

```
/******************************************************************
* FILE: bug2.c
* DESCRIPTION:
*   A "hello world" Pthreads program that dumps core.  Figure out why
and
*   then fix it - or else see the solution bug2fix.c.
* AUTHOR: 9/98 Blaise Barney
* LAST REVISED: 01/29/09
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS  8
#define ARRAY_SIZE      500000

void *Hello(void *threadid)
{
   double A[ARRAY_SIZE];
   int i;
   long tid;

   tid = (long)threadid;
   sleep(3);
   for (i=0; i<ARRAY_SIZE; i++)
   {
      A[i] = i * 1.0;
   }
   printf("%ld: Hello World!   %f\n", tid, A[ARRAY_SIZE-1]);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
pthread_t threads[NTHREADS];
size_t stacksize;
pthread_attr_t attr;
int rc;
long t;
pthread_attr_init(&attr);
pthread_attr_getstacksize (&attr, &stacksize);
printf("Thread stack size = %li bytes (hint, hint)\n",stacksize);
for(t=0;t<NTHREADS;t++){
  rc = pthread_create(&threads[t], NULL, Hello, (void *)t);
  if (rc){
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }
printf("Created %ld threads.\n", t);
pthread_exit(NULL);
```

```
}
```

Bug2fix.c

```
/******************************************************************
* FILE: bug2fix.c
* DESCRIPTION:
*   This is just one way to fix the "hello world" Pthreads program
that dumps
*   core.  Things to note:
*     - attr variable and its scoping
*     - use of the pthread_attr_setstacksize routine
*     - initialization of the attr variable with pthread_attr_init
*     - passing the attr variable to pthread_create
* AUTHOR: 9/98 Blaise Barney
* LAST REVISED: 01/29/09
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS  8
#define ARRAY_SIZE      500000
#define MEGEXTRA        1000000

pthread_attr_t attr;

void *Hello(void *threadid)
{
   double A[ARRAY_SIZE];
   int i;
   long tid;
   size_t mystacksize;

   tid = (long)threadid;
   sleep(3);
   for (i=0; i<ARRAY_SIZE; i++)
   {
      A[i] = i * 1.0;
   }
   printf("%ld: Hello World!   %f\n", tid, A[ARRAY_SIZE-1]);
   pthread_attr_getstacksize (&attr, &mystacksize);
   printf("%ld: Thread stack size = %li bytes \n", tid, mystacksize);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
pthread_t threads[NTHREADS];
size_t stacksize;
int rc;
long t;
pthread_attr_init(&attr);
stacksize = ARRAY_SIZE*sizeof(double) + MEGEXTRA;
pthread_attr_setstacksize (&attr, stacksize);
pthread_attr_getstacksize (&attr, &stacksize);
printf("Thread stack size = %li bytes (hint, hint)\n",stacksize);
for(t=0;t<NTHREADS;t++){
  rc = pthread_create(&threads[t], &attr, Hello, (void *)t);
    if (rc){
```

```
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
  }
printf("Created %ld threads.\n", t);
pthread_exit(NULL);
}
```

Bug3.c

```
/******************************************************************
* FILE: bug3.c
* DESCRIPTION:
*   This "hello world" Pthreads program demonstrates an unsafe
(incorrect)
*   way to pass thread arguments at thread creation. Compare with
hello_arg1.c.
* AUTHOR: Blaise Barney
* LAST REVISED: 01/29/09
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8

void *PrintHello(void *threadid)
{
   long taskid = (long)threadid;
   sleep(1);
   printf("Hello from thread %ld\n", taskid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int rc;
long t;

for(t=0;t<NUM_THREADS;t++) {
  printf("Creating thread %ld\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
   }

pthread_exit(NULL);
}
```

Bug4.c

```
/******************************************************************
* FILE: bug4.c
* DESCRIPTION:
*   This program demonstrates a condition variable
race/synchronization
```

```
*   problem. It resembles the condvar.c program. One possible solution
can
*   be found in bug4fix.c
* SOURCE: 07/06/05 Blaise Barney
* LAST REVISED: 01/29/09  Blaise Barney
****************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* Define and scope what needs to be seen by everyone */
#define NUM_THREADS  3
#define ITERATIONS 10
#define THRESHOLD 12
int count = 0;
double finalresult=0.0;
pthread_mutex_t count_mutex;
pthread_cond_t count_condvar;


void *sub1(void *t)
{
  int i;
  long tid = (long)t;
  double myresult=0.0;

  /* do some work */
  sleep(1);

  /* Lock mutex and wait for signal only if count is what is expected.
  Note that the pthread_cond_wait routine will automatically and
  atomically unlock mutex while it waits. Also, note that if THRESHOLD
  is reached before this routine is run by the waiting thread, the
  loop will be skipped to prevent pthread_cond_wait from never
  returning, and that this thread's work is now done within the mutex
  lock of count. */
  pthread_mutex_lock(&count_mutex);
  printf("sub1: thread=%ld going into wait. count=%d\n",tid,count);
  pthread_cond_wait(&count_condvar, &count_mutex);
  printf("sub1: thread=%ld Condition variable signal received.",tid);
  printf(" count=%d\n",count);
  count++;
  finalresult += myresult;
  printf("sub1: thread=%ld count now equals=%d myresult=%e. Done.\n",
         tid,count,myresult);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

void *sub2(void *t)
{
  int j,i;
  long tid = (long)t;
  double myresult=0.0;

  for (i=0; i<ITERATIONS; i++) {
    for (j=0; j<100000; j++)
      myresult += sin(j) * tan(i);
    pthread_mutex_lock(&count_mutex);
    finalresult += myresult;
    count++;
```

```
      /* Check the value of count and signal waiting thread when
      condition is reached.  Note that this occurs while mutex is
      locked.  */
    if (count == THRESHOLD) {
      printf("sub2: thread=%ld Threshold reached. count=%d.
",tid,count);
      pthread_cond_signal(&count_condvar);
      printf("Just sent signal.\n");
      }
    else {
      printf("sub2: thread=%ld did work. count=%d\n",tid,count);
      }
    pthread_mutex_unlock(&count_mutex);
    }
    printf("sub2: thread=%ld  myresult=%e. Done. \n",tid,myresult);
  pthread_exit(NULL);
}



int main(int argc, char *argv[])
{
  long t1=1, t2=2, t3=3;
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_condvar, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, sub1, (void *)t1);
  pthread_create(&threads[1], &attr, sub2, (void *)t2);
  pthread_create(&threads[2], &attr, sub2, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d threads. Final result=%e. Done.\n",
          NUM_THREADS,finalresult);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_condvar);
  pthread_exit (NULL);

}
```

Bug4fix.c

```
/******************************************************************
* FILE: bug4fix.c
* DESCRIPTION:
```

```
*   This is just one way to resolve the synchronization problem
demonstrated
*   by bug4.c. A check is made in sub1 to make sure the
pthread_cond_wait()
*   call is not made if the value of count is not what it expects. Its
work is
*   also placed after it is awakened, while count is locked.
* SOURCE: 07/06/05 Blaise Barney
* LAST REVISED: 01/29/09  Blaise Barney
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* Define and scope what needs to be seen by everyone */
#define NUM_THREADS  3
#define ITERATIONS 10
#define THRESHOLD 12
int count = 0;
double finalresult=0.0;
pthread_mutex_t count_mutex;
pthread_cond_t count_condvar;


void *sub1(void *t)
{
  int i;
  long tid = (long)t;
  double myresult=0.0;

  /* Lock mutex and wait for signal only if count is what is expected.
  Note that the pthread_cond_wait routine will automatically and
  atomically unlock mutex while it waits. Also, note that if THRESHOLD
  is reached before this routine is run by the waiting thread, the
  loop will be skippedto prevent pthread_cond_wait from never
  returning, and that this thread's work is now done within the mutex
  lock of count. */
  pthread_mutex_lock(&count_mutex);
  if (count < THRESHOLD) {
    printf("sub1: thread=%ld going into wait. count=%d\n",tid,count);
    pthread_cond_wait(&count_condvar, &count_mutex);
    printf("sub1: thread=%ld Condition variable signal
received.",tid);
    printf(" count=%d\n",count);
    /* do some work */
    sleep(1);
    count++;
    finalresult += myresult;
    printf("sub1: thread=%ld count now equals=%d myresult=%e.
Done.\n",
           tid,count,myresult);
    }
  else {
    printf("sub1: count=%d. Not as expected.",count);
    printf(" Probably missed signal. Skipping work and exiting.\n");
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

void *sub2(void *t)
```

```
{
  int j,i;
  long tid = (long)t;
  double myresult=0.0;

  for (i=0; i<ITERATIONS; i++) {
    for (j=0; j<100000; j++)
      myresult += sin(j) * tan(i);
    pthread_mutex_lock(&count_mutex);
    finalresult += myresult;
    count++;
    /*Check the value of count and signal waiting thread when
    condition is reached.  Note that this occurs while mutex is
    locked. */
    if (count == THRESHOLD) {
      printf("sub2: thread=%ld Threshold reached. count=%d.
",tid,count);
      pthread_cond_signal(&count_condvar);
      printf("Just sent signal.\n");
      }
    else {
      printf("sub2: thread=%ld did work. count=%d\n",tid,count);
      }
    pthread_mutex_unlock(&count_mutex);
    }
    printf("sub2: thread=%ld  myresult=%e. Done. \n",tid,myresult);
  pthread_exit(NULL);
}



int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_condvar, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, sub1, (void *)t1);
  pthread_create(&threads[1], &attr, sub2, (void *)t2);
  pthread_create(&threads[2], &attr, sub2, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d threads. Final result=%e. Done.\n",
          NUM_THREADS,finalresult);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_condvar);
  pthread_exit (NULL);
```

```
}
```

Bug5.c

```
/******************************************************************
* FILE: bug5.c
* DESCRIPTION:
*   A simple pthreads program that dies before the threads can do
their
*   work. Figure out why.
* AUTHOR: 07/06/05 Blaise Barney
* LAST REVISED: 01/29/09
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   int i;
   double myresult=0.0;
   printf("thread=%ld: starting...\n", threadid);
   for (i=0; i<1000000; i++)
      myresult += sin(i) * tan(i);
   printf("thread=%ld result=%e. Done.\n",threadid,myresult);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int rc;
long t;
for(t=0;t<NUM_THREADS;t++){
  printf("Main: creating thread %ld\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
  if (rc){
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }
printf("Main: Done.\n");
}
```

Bug6.c

```
/******************************************************************
* FILE: bug6.c
* DESCRIPTION:
*   This example demonstrates a race condition with a global variable
that
*   gives obviously wrong results. Figure out how to fix the problem -
or see
*   bug6fix.c for one solution. The dotprod_mutex.c example provides a
much
*   more efficient way of solving this problem than bug6fix.c (FYI).
```

```
* SOURCE: 07/06/05 Blaise Barney
* LAST REVISED: 01/29/09 Blaise Barney
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* Define global data where everyone can see them */
#define NUMTHRDS 8
#define VECLEN 100000
int *a, *b;
long sum=0;

void *dotprod(void *arg)
{
  /* Each thread works on a different set of data.
   * The offset is specified by the arg parameter. The size of
   * the data for each thread is indicated by VECLEN.
   */
   int i, start, end, offset, len;
   long tid = (long)arg;
   offset = tid;
   len = VECLEN;
   start = offset*len;
   end   = start + len;

/* Perform my section of the dot product */
   printf("thread: %ld starting. start=%d end=%d\n",tid,start,end-1);
   for (i=start; i<end ; i++)
      sum += (a[i] * b[i]);
   printf("thread: %ld done. Global sum now is=%li\n",tid,sum);

   pthread_exit((void*) 0);
}



int main (int argc, char *argv[])
{
long i;
void *status;
pthread_t threads[NUMTHRDS];
pthread_attr_t attr;

/* Assign storage and initialize values */
a = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));
b = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));

for (i=0; i<VECLEN*NUMTHRDS; i++)
  a[i]= b[i]=1;

/* Create threads as joinable, each of which will execute the dot
product
 * routine. Their offset into the global vectors is specified by
passing
 * the "i" argument in pthread_create().
 */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
for(i=0; i<NUMTHRDS; i++)
   pthread_create(&threads[i], &attr, dotprod, (void *)i);
```

```
pthread_attr_destroy(&attr);

/* Wait on the threads for final result */
for(i=0; i<NUMTHRDS; i++)
  pthread_join(threads[i], &status);

/* After joining, print out the results and cleanup */
printf ("Final Global Sum=%li\n",sum);
free (a);
free (b);
pthread_exit(NULL);
}
```

Bug6fix.c

```
/******************************************************************
* FILE: bug6fix.c
* DESCRIPTION:
*   This solution uses a mutex variable to protect the global sum
while each
*   thread updates it. A much more efficient method would be that used
in the
*   dotprod_mutex.c example.
* SOURCE: 07/06/05 Blaise Barney
* LAST REVISED: 01/29/09 Blaise Barney
******************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* Define global data where everyone can see them */
#define NUMTHRDS 8
#define VECLEN 10000
pthread_mutex_t mutexsum;
int *a, *b;
long sum=0.0;

void *dotprod(void *arg)
{
  /* Each thread works on a different set of data.
   * The offset is specified by the arg parameter. The size of
   * the data for each thread is indicated by VECLEN.
   */
   int i, start, end, offset, len;
   long tid;
   tid = (long)arg;
   offset = tid;
   len = VECLEN;
   start = offset*len;
   end   = start + len;

/* Perform my section of the dot product */
   printf("thread: %ld starting. start=%d end=%d\n",tid,start,end-1);
   for (i=start; i<end ; i++) {
      pthread_mutex_lock(&mutexsum);
      sum += (a[i] * b[i]);
      pthread_mutex_unlock(&mutexsum);
      }
   printf("thread: %ld done. Global sum now is=%li\n",tid,sum);
```

```
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[])
{
long i;
void *status;
pthread_t threads[NUMTHRDS];
pthread_attr_t attr;

/* Assign storage and initialize values */
a = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));
b = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));

for (i=0; i<VECLEN*NUMTHRDS; i++)
  a[i]=b[i]=1;

/* Initialize mutex variable */
pthread_mutex_init(&mutexsum, NULL);

/* Create threads as joinable, each of which will execute the dot
product
 * routine. Their offset into the global vectors is specified by
passing
 * the "i" argument in pthread_create().
 */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
for(i=0;i<NUMTHRDS;i++)
   pthread_create(&threads[i], &attr, dotprod, (void *)i);

pthread_attr_destroy(&attr);

/* Wait on the other threads for final result */

for(i=0;i<NUMTHRDS;i++) {
  pthread_join(threads[i], &status);
  }
/* After joining, print out the results and cleanup */
printf ("Final Global Sum=%li\n",sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

Apéndice E.

Referencias e hiperenlaces

1. Manuel Jesús Martín Gutierrez. 2011. Módulo Empresarial para Java Path Finder
2. Ubuntu 11.04
   http://www.ubuntu.com/ubuntu
3. JDK 6
   http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u2-download-1377129.html
4. Netbeans IDE
   http://netbeans.org
5. Glassfish
   http://glassfish.java.net/
6. Valgrind
   http://valgrind.org/
7. Wikipedia; "Formal Verification"
   http://en.wikipedia.org/wiki/Formal_verification
8. Java Path Finder
   http://babelfish.arc.nasa.gov/trac/jpf
9. MyGCC
   http://mygcc.free.fr/
10. Oracle; Java EE Documentation
    http://www.oracle.com/technetwork/java/javaee/documentation/index.html
11. Oracle; "EJBs Specifications"
    http://www.oracle.com/technetwork/java/docs-135218.html
12. Wikipedia; "Enterprise JavaBeans"
    http://es.wikipedia.org/wiki/Enterprise_JavaBeans
13. Oracle; "Java SE Documentation"
    http://www.oracle.com/technetwork/java/javase/documentation/index.html
14. Oracle; "Java EE 6 Tutorial"
    http://docs.oracle.com/javaee/6/tutorial/doc/
15. Sun Microsystems; "JavaMail API"

http://java.sun.com/products/javamail/javadocs/javax/mail/package-summary.html

16. Sun Microsystems; "java.util.zip API" http://download.oracle.com/javase/6/docs/api/java/util/zip/package-summary.html
17. Oracle; "BEA Weblogic server" http://www.oracle.com/technetwork/middleware/Weblogic/overview/index.html
18. JBoss; "JBoss application server" http://www.jboss.org/jbossas
19. IBM; "IBM WebSphere" http://www-01.ibm.com/software/websphere/#
20. Borland; "Borland AppServer" http://www.borland.com/us/products/appserver/index.html
21. Sun Microsystems; "JMS Specifications" http://java.sun.com/products/jms/jms1_0_2-spec.pdf
22. Apache Software Foundation; "Jakarta Commons HTTPClient Tutorial" http://hc.apache.org/httpclient-3.x/tutorial.html
23. Stanford; "Oreilly MultipartParser API" http://www.stanford.edu/group/coursework/docsTech/oreilly/com.oreilly.servlet.multipart.MultipartParser.html
24. El País; Oracle compra Sun Microsystems http://www.elpais.com/articulo/internet/Oracle/adquiere/Sun/Microsystems/5710/millones/elpeputec/20090420elpepunet_3/Tes
25. Valgrind; User Manual http://valgrind.org/docs/manual/manual.html
26. POSIX; Wikipedia http://es.wikipedia.org/wiki/POSIX
27. Javascript; Wikipedia http://es.wikipedia.org/wiki/Javascript
28. Ejercicios POSIX https://computing.llnl.gov/tutorials/pthreads/exercise.html
29. Sun Microsystems; API JMS http://docs.oracle.com/cd/E17802_01/products/products/jms/javadoc-102a/index.html
30. Sun Microsystems; API LinkedList http://docs.oracle.com/javase/1.4.2/docs/api/java/util/LinkedList.html
31. Sun Microsystems; API StringTokenizer http://docs.oracle.com/javase/1.4.2/docs/api/java/util/StringTokenizer.html
32. Sun Microsystems; API Swing Timer http://docs.oracle.com/javase/1.4.2/docs/api/javax/swing/Timer.html
33. Sun Microsystems; API AWT Event

http://docs.oracle.com/javase/1.5.0/docs/api/java/awt/event/package-summary.html

34. Wikipedia; GCC
http://es.wikipedia.org/wiki/GNU_Compiler_Collection

35. GCC, the GNU Compiler Collection
http://gcc.gnu.org/